Introducing OBJ*

Joseph A. Goguen[†], Timothy Winkler[‡], José Meseguer[‡],
Kokichi Futatsugi[§], and Jean-Pierre Jouannaud[¶]

Abstract

This is an introduction to the philosophy and use of OBJ, emphasizing its operational semantics, with aspects of its history and its logical semantics. Release 2 of OBJ3 is described in detail, with many examples. OBJ is a wide spectrum first-order functional language that is rigorously based on (order sorted) equational logic and parameterized programming, supporting a declarative style that facilitates verification and allows OBJ to be used as a theorem prover.

Order sorted algebra provides a notion of *subsort* that rigorously supports multiple inheritance, exception handling and overloading. *Parameterized programming* gives powerful support for design, verification, reuse, and maintenance, using two kinds of module: *objects* to encapsulate executable code, and in particular to define abstract data types by initial algebra semantics; and (loose) *theories* to specify both syntactic and semantic properties of modules. Each kind of module can be parameterized, where actual parameters may be modules. For parameter instantiation, a *view* binds the formal entities in an interface theory to actual entities in a module, and also asserts that the target module satisfies the semantic conditions of the interface theory. *Module expressions* allow complex combinations of already defined modules, including sums, instantiations, and transformations; moreover, evaluating a module expression actually constructs the described software (sub)system from the given components. *Default views* can greatly reduce the effort of instantiating modules, by allowing obvious correspondences to be left out. We argue that first-order parameterized programming includes much of the power of higher-order programming, in a form that is often more convenient.

Although OBJ executable code normally consists of equations that are interpreted as rewrite rules, OBJ3 objects can also encapsulate Lisp code, e.g., to provide efficient built-in data types, or to augment the system with new capabilities; we describe the syntax of this facility, and provide some examples. In addition, OBJ provides rewriting modulo associative, commutative and/or identity equations, as well as user-definable evaluation strategies that allow lazy, eager, and mixed evaluation strategies on an operator-by-operator basis; memoization is also available on an operator-by-operator basis. In addition, OBJ3 supports the application of equations one at a time, either forwards or backwards; this is needed for equational theorem proving. Finally, OBJ provides user-definable mixfix syntax, which supports the notational conventions of particular application domains.

1 Introduction

This paper motivates and describes the use of OBJ, based on Release 2.04 of the OBJ3 system. OBJ3 is the latest in a series of OBJ systems, each of which has been rigorously based upon equational logic; however, the semantic basis of OBJ is not developed here in full detail. The OBJ3 system is implemented in Common Lisp, and is based on ideas from order sorted algebra and parameterized programming. OBJ3 provides mixfix syntax, flexible subsorts, parameterized modules, views, and rewriting modulo associativity, commutativity, and/or identity. With its module database and its ability to incorporate Lisp code, this provides a very flexible and extensible environment that is convenient for specification and rapid prototyping, as well as for building new systems, such as experimental languages and theorem proving environments. For example,

*The research reported in this paper has been supported in part by: Office of Naval Research Contracts N00014-85-C-0417, N00014-86-C-0450, and N00014-90-C-0086; NSF Grant CCR-8707155; grants from the Science and Engineering Research Council; a gift from the System Development Foundation; and a grant from Fujitsu Laboratories.

[†]Department of Computer Science & Engineering, University of California at San Diego, La Jolla CA 92093-0114, USA.
[‡]SRI International, Menlo Park CA 94025, USA.
[§]Japan Advanced Institute of Science and Technology, 1-1 Asahidai, Tatsunokuchi, Ishakawa, Japan.
[¶]Université de Paris-Sud, 91405 Orsay, France.

OBJ3 has been used for building FOOPS, an object oriented specification and programming system [74, 87], the Eqlog system [71, 72, 25] for equational logic (or relational) programming, OOZE [3], an object oriented specification language influenced by Z [142], the 2OBJ metalogical framework theorem prover [81], and TOOR [130], a system for tracing requirements.

OBJ has been used for many applications, including debugging algebraic specifications [77], rapid prototyping [69], defining programming languages in a way that directly yields an interpreter (see Appendix Section C.2, as well as [79] and some elegant work of Peter Mosses [120, 121]), specifying software systems (e.g., the GKS graphics kernel system [28], an Ada configuration manager [40], the MacIntosh QuickDraw program [126], and OBJ itself [20]), hardware specification, simulation, and verification (see [144] and Section 4.8), specification and verification of imperative programs [66], specification of user interface designs [60, 58], and theorem proving [53, 66, 59]; several of these were done under a UK government grant. OBJ has also been combined with Petri nets, thus allowing structured data in tokens [5], and was used to verify compilers for parallel programming languages in the ESPRIT sponsored PROCOS project [137, 138]. In addition, OBJ serves as a programming language for the massively parallel Rewrite Rule Machine, which executes rewrite rules directly [63, 152, 106, 75, 107, 2, 54, 1]; in fact, given equal silicon floorspace and development effort, OBJ on such a machine could out-perform a conventional language on a conventional machine, because of the direct concurrent execution of rewrite rules. Some examples using OBJ3 for theorem proving and hardware verification from [51] and [59] are given in Appendix Section C.4. In [66], OBJ3 is used for teaching the semantics of imperative programming languages, and all the proofs in [66] are actually executable OBJ3 programs; see [109] for a more detailed discussion of the educational uses of OBJ.

1.1 A Brief History of OBJ

OBJ was designed in 1976 by Goguen [43], using "error algebras" to extend algebraic abstract data type theory with error handling and partial functions; this first design also used ideas from Clear [8, 10] for parameterized modules, thus giving birth to parameterized programming. The first implementations of OBJ were done from 1977 to 1979 at UCLA by Joseph Tardo and Joseph Goguen. OBJ0 [45] was based on unsorted equational logic, while OBJT used error algebras plus an "image" construct for parameterization [148, 82]. David Plaisted implemented OBJ1, building on OBJT during 1982–83 at SRI, based on theoretical and design work carried out with Joseph Goguen and José Meseguer; improvements of OBJ1 over OBJT included rewriting modulo associativity and/or commutativity, hash coded memo functions, the use of theories with loose semantics as well as objects with initial semantics, and new interactive features [77] which made the system more convenient for users.

OBJ2 [35, 36] was implemented using parts of OBJ1 during 1984-85 at SRI by Kokichi Futatsugi and Jean-Pierre Jouannaud, following a design in which José Meseguer and Joseph Goguen also participated, based on order sorted algebra [44, 76, 62, 73, 141] rather than error algebra; also, OBJ2 provided Clear-like parameterized modules, theories, and views, although not in full generality. Another influence on OBJ3's design and implementation was the HISP system [38, 39, 30] of Kokichi Futatsugi.

OBJ3 was first developed at SRI by Timothy Winkler, José Meseguer, Joseph Goguen, Claude and Hélène Kirchner, and Aristide Megrelis; Release 2 was developed at SRI by Timothy Winkler, Patrick Lincoln, José Meseguer, and Joseph Goguen, and later extended by Winkler and Goguen at the Programming Research Group of the Oxford University Computing Lab; version 2.04 includes further small bug fixes made at Oxford. Although the syntax of OBJ3 is close to that of OBJ2, it has a different implementation based on a simpler approach to order sorted rewriting [101], and it also provides much more sophisticated parameterized programming. OBJ2 and OBJ3 can be seen as implementations of Clear [8, 10], where the chosen logic is order sorted equational logic.

Other implementations of OBJ1 include UMIST-OBJ from the University of Manchester Institute of Science and Technology [20], Abstract Pascal from the University of Manchester [105], and MC-OBJ from the University of Milan [14]; the first two are written in Pascal and the third in C. In addition, there is a Franz Lisp OBJ2 done at Washington State University [143]. UMIST-OBJ has been made available as a proprietary software product from Gerrard Software, under the name ObjEx.

OBJ has been extended in many directions, including logic (or relational) programming (the Eqlog system [71, 72, 25]), object oriented programming (the FOOPS system [74, 87]), object oriented specification (OOZE [3]), requirements tracing (TOOR [130]), higher-order functional programming [97, 110], and LOTOS-style specification for communication protocols [127, 128].

Recent developments within the OBJ community include CafeOBJ, Maude, and CASL. CafeOBJ [26, 27] is being built at the Japan Institute of Science and Technology under the direction of Prof. Kokichi Futatsugi; it extends OBJ3 with hidden algebra [55, 80, 68, 67] for behavioral specification, and with rewriting logic [113, 114] for applications programming. Maude [112, 16] is being built at SRI International under the direction of Dr. José Meseguer; it extends OBJ with rewriting logic, and has been successfully used for metaprogramming, reflection, and algorithm implementation [17, 18]. CASL [19] is being developed by a European consortium called CoFI, whose members include Maura Cerioli, Till Mossakowsky, Peter Mosses, Don Sannella, Bernd Krieg-Bruckner, Michel Bidoit, and Andre Tarlecki; this system is loosely based on OBJ ideas, including parameterized programming, but is more oriented towards specification than verification and prototyping, and is intended to support experimentation with a variety of logics. As this document is being finished, a system called Kumo is being installed at UCSD. Kumo is a proof assistant for first order hidden logic, that also generates websites to document the proofs obtained [65, 64, 78, 60]; it greatly extends the theorem proving power of OBJ, in the logics supported, in the level of automation obtained, and in the user interface, including proof documentation.

1.2 A Brief Summary of Parameterized Programming

OBJ has three kinds of entity at its top level: objects, theories, and views. An **object** encapsulates executable code, while a **theory** defines properties that may (or may not) be satisfied by another object or theory. Both objects and theories are **modules**. A **view** is a *binding* of the entities declared in some theory to entities in some other module, and also an assertion that the other module satisfies the properties declared in the theory. Theories and views are found in no other implemented language with which we are familiar; however, Standard ML has been influenced by this approach.

Modules can import other previously defined modules, and therefore an OBJ program is conceptually a *graph* of modules. Modules have **signatures** that introduce new sorts[1] and new operators[2] among both new and old sorts. In addition, **variables** with declared sorts are introduced. **Terms** are built up from variables and operators, respecting their sort declarations. Modules can be parameterized, and parameterized modules use theories to define both the syntax and the *semantics* of their interfaces. Views indicate how to instantiate a parameterized module with an actual parameter.

This kind of module composition is, in practice, more powerful than the purely functional composition of traditional functional programming, because a single module instantiation can compose together many different functions all at once, in complex ways. For example, a parameterized complex arithmetic module `CPXA` can easily be instantiated with any of several real arithmetic modules as actual parameter:

- single precision reals, `CPXA[SP-REAL]`,
- double precision reals, `CPXA[DP-REAL]`,
- multiple precision reals, `CPXA[MP-REAL]`,

where `SP-REAL`, `DP-REAL`, and `MP-REAL` are modules for single-precision, double-precision, and multiple precision floating point arithmetic. Each instantiation may involve substituting dozens of functions into the definitions of dozens of other functions. While something similar is possible in higher-order functional programming by coding up modules as records, it seems much less natural, particularly if this encoding also specifies the semantics of the interface of `CPXA`. Furthermore, parameterized programming allows the logic to remain first-order, so that understanding and verification can be simpler. Section 4.7 shows that many typical higher-order functional programming techniques can be implemented with OBJ parameterized modules, often with essentially the same flexibility and with greater clarity, while Section 4.8 shows that a typical application of higher-order functions, namely hardware verification, is readily captured in the simpler formalism.

1.3 An Overview of this Document

OBJ3's top level accepts declarations (for objects, theories and views), and commands to `reduce` terms, to `show` various status and structural information, to `set` various conditions, and to `do` various actions. A **reduction** evaluates a given term with respect to a given object, and OBJ supports reduction modulo associativity, commutativity, and/or identity. Section 2 describes objects, Section 4.1 describes theories, Section 4.2 describes views, and Section 2.3.1 describes reduction. The `show`, `set` and `do` commands are discussed in various places, but are collected in Appendix A. OBJ's approach to imported modules and to built-in sorts and operators is described in Section 3, with many further details of the latter in Appendix

[1] Here and hereafter, we generally use the word "sort" instead of "type" because of the very many different meanings that have been assigned to the word "type."

[2] Although we generally use the word "operator," in this paper it usually means "function" because we are dealing with a functional language.

D. Built-ins can be useful, for example, in adding new built-in data types to OBJ, or in extending or modifying OBJ in various other ways. Section 5 discusses how to apply rewrite rules one at a time, forwards or backwards, which is needed for theorem proving. Section 6 tries to discuss what OBJ is and is not. Appendix A gives some hints on how to use OBJ, Appendix B gives OBJ3's syntax, and Appendix C gives many examples. We have tried to give a fairly comprehensive bibliography of papers that use OBJ.

1.4 Acknowledgements

We wish to thank: Professor Rod Burstall for his extended collaboration on Clear and its foundations, which inspired the parameterization mechanism of OBJ; Professor David Plaisted for his many suggestions about the design of OBJ, and for his implementation of OBJ1; Dr. Joseph Tardo for his pioneering and courageous implementation of OBJT; Drs. James Thatcher, Eric Wagner and Jesse Wright for their initial collaboration on abstract data types; Dr. Peter Mosses for his valuable suggestions and his efforts to use a very early version of OBJ3; Dr. Victoria Stavridou for her use of OBJ3 in hardware specification and verification; Dr. Claude Kirchner for his work on the pattern matching routines and rule generation for OBJ3; Dr. Hélène Kirchner for her work on the rule generation for OBJ3; Dr. Adolfo Socorro for help checking the details of OBJ3 syntax; Dr. Patrick Lincoln for help with the routines for rewriting modulo equations used in Release 2 of OBJ3; Mr. Aristide Megrelis for his work on the OBJ3 parser; and several generations of students at Oxford and UCSD for their feedback on using OBJ3 in courses on theorem proving, and on the semantics of programming languages [57]. Much of this paper is based upon [47, 52, 86] and [35].

2 Objects

The most important OBJ unit is the **object**[3], which encapsulates executable code. Syntactically, an object begins with the keyword `obj` and ends with `endo`[4]. The name of the object occurs immediately after the `obj` keyword; following this comes `is`, and then the **body** of the object. For unparameterized objects, the name is a simple identifier, such as `STACK-OF-INT`, `PHRASE` or `OBJ14`. Parameterized objects have an interface specification in place of a simple name, as discussed in Section 4. Schematically, the form is

```
obj ⟨ModId⟩ is
  .....
endo
```

where ⟨*ModId*⟩ is a metasyntactic symbol for a module identifier, by convention all upper case, possibly including special characters; however, this convention is not enforced, and any character string not containing blanks (i.e., spaces) or special characters can be used. OBJ keywords are lower case.

2.1 Strong Sorting and Subsorts

We believe that languages should have strong but flexible "type systems." Among the advantages of "strong typing," which of course we call **strong sorting**, are: to catch meaningless expressions before they are executed; to separate logically and intuitively distinct concepts; to enhance readability by documenting these distinctions; and, when the notion of subsort is added, to support multiple inheritance, overloading (a form of subsort polymorphism), coercions, multiple representations, and error handling, without the confusion, and lack of semantics, found in many programming languages (see [73] for a more detailed discussion of these issues). In particular, overloading can allow users to write simpler expressions, because context can often determine which possibility is intended. Of course, strong sorting may require additional declarations, but with a modern editor, it is little trouble to insert declarations, and many could even be generated automatically.

Ordinary unsorted logic offers the dubious advantage that anything can be applied to anything; for example,

$$\texttt{first-name(not(age(3 * false))) iff } 2^{\texttt{birth-place(temperature(329))}}$$

[3] Objects in this sense are not very closely related to objects in the sense of object-oriented programming; rather, they provide executable algebraic specifications for abstract data types.

[4] OBJ3 has the uniform convention that ending keywords can be of the form "end<x>" where "<x>" is the first letter, or first two letters, of the corresponding initial keyword. The initial keyword spelled backwards, as in `jbo`, is an archaic form for some keywords preserved from earlier versions of OBJ.

is a well formed expression. Although beloved by Lisp and Prolog hackers, unsorted logic is too permissive. Unfortunately, the obvious alternative, many sorted logic, is too restrictive, because it does not support overloaded function symbols, such as `_+_` for integer, rational, and complex numbers. Moreover, strictly speaking, an expression like `(-4 / -2)!` does not even parse (assuming that factorial only applies to natural numbers), because `(-4 / -2)` looks to the parser like a rational rather than a natural. In Section 2.3.3 below, we show that order sorted algebra with *retracts* provides sufficient expressiveness, while still banishing truly meaningless expressions.

Let us now be specific. Sorts are declared in OBJ3 with the syntax

`sorts` ⟨*SortIdList*⟩ `.`

where ⟨*SortIdList*⟩ is a list of ⟨*SortId*⟩s, as in

```
sorts Nat Int Rat .
```

When there is just one sort, it may be more fluent to write

`sort` ⟨*SortId*⟩ `.`

However, `sort` and `sorts` are actually synonymous.

Warning: Sort declarations must be terminated with a blank followed by a period.

Order sorted algebra, sometimes abbreviated **OSA** in the following, is designed to handle cases where things of one sort are also of another sort (e.g., all natural numbers are also rational numbers), and where operators or expressions may have several different sorts. The essence of order sorted algebra is to provide a *subsort* partial ordering among sorts, and to interpret it semantically as subset inclusion among the carriers of models; for example, `Nat < Rat` means that $M_{\mathtt{Nat}} \subseteq M_{\mathtt{Rat}}$, where M is a model, and M_s is its set of elements of sort s. (Note that OBJ uses `<` instead of $\leq$ simply for typographical convenience.) OSA also supports **multiple inheritance**, in the sense that a given sort may have *more than one* distinct supersort.

Although many sorted algebra has been quite successful for the theory of abstract data types, it can produce some very awkward code in practice, primarily due to difficulties in handling erroneous expressions, such as dividing by zero in the rationals, or taking the top of an empty stack. In fact, there *is no* really satisfying way to define either rationals or stacks with (unconditional) many sorted algebra: [83] and [29] contain some examples which show just how awkward things can get, and [73] actually proves that certain kinds of specifications cannot be expressed at all in many sorted equational logic.

OSA overcomes these obstacles with its subsorts and overloaded operators, and it allows functions to be total that would otherwise have to be partial, by restricting them to a subsort. Two pleasant facts are that OSA is only slightly more difficult than many sorted algebra, and that essentially all results generalize without difficulty from the many sorted to the order sorted case. Although this paper omits the technical details, OSA is a rigorous mathematical theory. OSA was originally suggested by Goguen in 1978 [44], and is further developed in [76] and [73]; some alternative approaches have been given by Gogolla [41, 42], Mosses [122], Poigne [133, 134], Reynolds [135], Smolka *et al.* [140, 141], Wadge [150], and others. A survey as of 1993 appears in [61], along with some new generalizations. Meseguer has recently proposed a new generalization, called membership equational logic [115].

OBJ3 directly supports *subsort polymorphism*, which is operator overloading that is consistent under subsort restriction (this is further discussed in Section 2.2). By contrast, languages like ML [91], Hope [13] and Miranda [149] support *parametric polymorphism*, following ideas of Strachey [147] as further developed by Milner [119]. OBJ3's parameterized modules also provide a parametric capability, but instantiations are determined by views, rather than by unification; see Section 4.7 for further discussion.

The basic syntax for a subsort declaration in OBJ3 is

`subsort` ⟨*Sort*⟩ `<` ⟨*Sort*⟩ `.`

which means that the set of things having the first ⟨*Sort*⟩[5] is a subset (not necessarily proper) of the things having the second ⟨*Sort*⟩. Similarly,

`subsorts` ⟨*SortList*⟩ `<` ⟨*SortList*⟩ `< ... .`

[5]Note that ⟨*Sort*⟩s differ from ⟨*SortId*⟩s in allowing qualification by module name; see Appendix B for details of the notation for syntax that is used in this paper.

means that each sort in the first ⟨*SortList*⟩ is a subsort of each sort in the second ⟨*SortList*⟩, and so on. (Actually `subsort` and `subsorts` are synonyms.)

Warning: The elements of each list must be separated by blanks, and the declaration must be terminated with a blank followed by a period. OBJ3 complains if any sort in a subsort declaration doesn't appear in a previous sort declaration, or if there are cycles in the graph of the subsort relation. Subsort cycles may produce strange behavior.

2.2 Operator and Expression Syntax

We believe it is worth some extra implementation effort and processing time to support syntax that is as flexible, as informative, and as close as possible to users' intuitions and standard usage in particular problem domains. Thus, users of OBJ can define any syntax they like for operators, including prefix, postfix, infix, and most generally, **mixfix**; this is similar to ECL [15]. Obviously, there are many opportunities for ambiguity in parsing such a syntax. OBJ's convention is that a term is **well formed** if and only if it has exactly one parse, or more precisely, a unique parse of *least sort*; it is intended that the parser give information about difficulties that it encounters, including multiple parses of least sort.

Warning: Due to the treatment of user-supplied operator precedence (see Section 2.4.3), the parser in Release 2 of OBJ3 may sometimes fail to find a parse, even though an unambiguous parse exists. This can usually be repaired by adding parentheses.

Let us now discuss operator syntax. The argument and value sorts of an operator are declared at the same time that its **syntactic form** is declared. There are two kinds of syntactic form declaration in OBJ. We call the first kind the **standard form**, because it defines the parenthesized-prefix-with-commas syntax that is standard in mathematics. For example,

```
op push : Stack Int -> Stack .
```

declares syntax for terms of the form `push(X,Y)` of sort `Stack`, where `X` has sort `Stack` and `Y` has sort `Int`. If the top operator of a term has standard syntactic form, then its arguments (i.e., its first level subterms) must be separated by commas and be enclosed within a top level matching pair of parentheses, for the entire term to be well formed. OBJ3's syntax for a standard operator declaration is

`op` ⟨*OpForm*⟩ `:` ⟨*SortList*⟩ `->` ⟨*Sort*⟩ `.`

where ⟨*OpForm*⟩ is a nonempty string of characters, possibly consisting of multiple (blank-separated) tokens. Operators in standard form should not include the underbar character, "`_`" (some further syntactic requirements for ⟨*OpForm*⟩ are discussed below).

Warning: An operator declaration must be terminated with a blank followed by a period[6], and all of the sorts (and operators — see the discussion of `id` declarations below) used in it must have been previously declared.

The second kind of OBJ syntax for operator declarations is called **mixfix form**, and it allows declaring arbitrary mixfix syntax. This kind of declaration uses place-holders, indicated by an underbar character, to indicate where arguments should appear; the rest of the operator form consists of the keywords associated with the operator. For example, the following is a prefix declaration for `top` as used in terms like `top push(S,5)`:

```
op top_ : Stack -> Int .
```

Similarly, the "outfix" form of the singleton set formation operator, as in `{ 4 }`, is declared by

```
op {_} : Int -> Set .
```

and the infix form for addition, as in `2 + 3`, is given by

```
op _+_ : Int Int -> Int .
```

while a mixfix declaration for conditional is

[6] An exception is if the last character is a left bracket, "`]`", which will occur if there are attributes (see Section 2.4).

```
op if_then_else_fi : Bool Int Int -> Int .
```

Between the `:` and the `->` in an operator declaration comes the **arity** of the operator, and after the `->` comes its **value sort** (sometimes called "co-arity"); the ⟨arity, value sort⟩ pair is called the **rank** of the operator. The general syntax for mixfix form operator declarations is

op ⟨*OpForm*⟩ : ⟨*SortList*⟩ -> ⟨*Sort*⟩ .

where ⟨*OpForm*⟩ is a non-empty string of characters, possibly consisting of multiple (blank-separated) tokens, possibly including blanks and matching pairs of parentheses. Blanks in the form have no effect when they are contiguous to an underbar. The following shows a form with a blank:

```
op _is in_ : Int IntSet -> Bool .
```

Warning: A mixfix operator form should be neither entirely blank, nor consist of just one underbar. Also, it must contain *exactly* as many underbars as there are sorts in its arity.

The entire ⟨*OpForm*⟩ of an operator can be enclosed in parentheses. This can be used to avoid syntactic ambiguity. For example, in the following declaration for division of rational numbers by non-zero rationals,

```
op (_:_) : Rat NzNat -> Rat .
```

failure to enclose the operator name in parentheses could cause the first "`:`" to be erroneously treated as the delimiter for the ⟨*SortList*⟩ of the declaration. Such enclosing parentheses are not considered part of the ⟨*OpForm*⟩, but rather provide a way to avoid this kind of syntactic ambiguity. The rule is that if the first token encountered after the "`op`" in an operator declaration is a left parenthesis that is matched by a right parenthesis before the delimiting "`:`", then these parentheses are interpreted as delimiters, rather than as part of the ⟨*OpForm*⟩. This does not preclude using parentheses as tokens in the syntax of an operator. However, the first token in the syntax of an operator should never be a left parenthesis. For example, one can declare an "apply" operator in a data type of lambda expressions with the syntax

```
op _(_) : Lambda Lambda -> Lambda .
```

Constant declarations have no underbars and have empty arity. For example,

```
op true : -> Bool .
```

Operators with the same rank but different forms can be declared together using the keyword `ops`; for example,

```
ops zero one : -> S .
ops (_+_) (_*_) : S S -> S .
```

The parentheses are required in the second case, to indicate the boundary between the two forms.

Warning: `op` and `ops` are *not* synonymous.

Here is a simple example illustrating some of the syntax given so far; it defines strings of bits.

```
obj BITS is
  sorts Bit Bits .
  ops 0 1 : -> Bit .
  op nil : -> Bits .
  op _._ : Bit Bits -> Bits .
endo
```

A typical term over the syntax declared in this object is `0 . 1 . 0 . nil`.

Warning: The period character, "`.`", is special in OBJ3, because it is used to terminate operator and equation declarations, among other constructions (although it is not required in simple situations where the input is self-delimiting). Sometimes you may need to enclose terms in parentheses to prevent an internal period from being interpreted as a final period. Tokens, such as "`.`", that are used to delimit syntactic units only function as delimiters when not enclosed in parentheses.

The `parse` command, with syntax

```
parse ⟨Term⟩ .
```

can be used at the top level of OBJ3, and also inside modules, to check the parsing of terms. It causes a fully parenthesized form of ⟨*Term*⟩ to be printed with its sort, provided it can be parsed.

When parsing fails, the system gives some diagnostic information that may help to discover the problem. For example,

```
red in INT : 1 + 10 div 2 .
```

produces the following

```
No successful parse for the input:
1 + 10 div 2
partial descriptions:
 _ _ _ div _
 1 + 10 _[div]_ 2
```

Two different "partial descriptions" are given of problematic tokens. In the first, the problematic tokens are replaced by "`_`". In the second, all tokens are displayed, but the problematic ones are enclosed within a matching "`_[`" and "`]_`" pair. This information can be very useful in detecting misspelled variable names and operator tokens. For example, the token "`div`" above should have been "`quo`". Tokens are considered problematic if they do not appear in any partial parse, where partial parses are generated by considering the prefixes of each suffix of the input string. In general, this only gives approximate information. For example, "`1 + 10`" is parsed as a prefix in the above expression, which may not be what the user intended.

When strong sorting is not sufficient to prevent ambiguity in a term that uses overloaded operators and subsorts, then **qualification** notation can be useful. For example, to distinguish the bit `0` in the object `BIT` above from the natural number zero, one can write `0.Bit` and `0.Nat`. Sort qualifiers can also be applied to mixfix operators; for example, `(X is in Set1 or Set2).Nat` could be used to distinguish a natural number valued operator `is in` (as used for bags) from a truth valued operator `is in` (as used for sets); the parentheses are not optional in this case.

Sometimes sort qualification doesn't work, but module qualification does, because different instances of the operator syntax have been introduced in different modules. For example, one might write `(when X is in Set2 do Act25).LANG`, where `LANG` is the name of a module. Sort names can also be qualified by module names, as in `Nat.NAT, Elt.X, Elt.Y`, etc. Qualification by module can be distinguished from qualification by sort as long as distinct names are involved, which happens naturally by using the convention suggested in this paper, that module names are all upper case, while sort names only have an upper case letter at the beginning (but note that this convention is not enforced by OBJ3).

Warning: Complete module names must be used, and the complete names of modules that have been created by evaluating module expressions can be surprisingly long; see Section 4.5.

Because OBJ3 is based on order sorted algebra, it supports **overloading**, so that the same operator symbol can have several different ranks. For example, addition might have the following declarations:

```
op _+_ : Nat Nat -> Nat .
op _+_ : Rat Rat -> Rat .
```

When the arity sorts of one operator declaration are less than those of another for the same function symbol, then in the models, the function interpreting the operator with smaller arity is the *restriction* to the smaller arity of the function interpreting the operator with larger arity. For example, the natural number addition of natural numbers should yield the same result as the rational number addition of the same natural numbers, provided we have declared `Nat < Rat`.

The **signature** of a module consists of the sorts, subsort relation, and operators available in it, where each operator has a form, arity, and value sort. Under the natural assumption of regularity, each order sorted term has a well defined least sort, where a signature is **regular** iff for any operator $f\colon w \to s$ and any $w' \leq w$ there is a least rank $\langle w'', s''\rangle$ among all $f\colon w'' \to s''$ satisfying $w' \leq w''$, where the ordering of ranks $\langle w, s\rangle$ is pointwise [62, 76]. A signature is **coherent** if each connected component of the set of sorts ordered by the subsort ordering has a top element, where the connected components are the equivalence classes under the equivalence relation obtained from the relation

sRs' iff $s \leq s'$ or $s' \leq s$

by closure under transitivity. To guarantee that OBJ3 works correctly, all signatures should be regular and coherent, and each connected component should have a top element; however, OBJ3 does not check these assumptions.

Using subsorts, and representing list concatenation by juxtaposition, we can now give a somewhat better representation for bit strings than that above:

```
obj BITS1 is
  sorts Bit Bits .
  subsorts Bit < Bits .
  ops 0 1 : -> Bit .
  op __ : Bit Bits -> Bits .
endo
```

A typical term using this syntax is `0 1 0` .

2.3 Equations and Semantics

We now turn to semantics. OBJ has both an abstract denotational semantics based on order sorted algebra, and a more concrete operational semantics based on order sorted term rewriting. The semantics of an object is determined by its equations. Equations are written declaratively, and are interpreted operationally as rewrite rules, which replace substitution instances of left sides by the corresponding substitution instances of right sides. Operational and denotational semantics are discussed in the next two subsections.

The following is a rather typical equation,

```
eq M + s N = s(M + N) .
```

where `M` and `N` are *variable symbols*, while `+` and `s` are operator symbols. They keyword "`eq`" introduces the equation, and the equality symbol "`=`" separates its left and right sides.

The syntax for declaring variables is

`vars` ⟨*VarIdList*⟩ `:` ⟨*Sort*⟩ `.`

where the variable names are separated by blanks. For example,

```
vars L M N : Nat .
```

The keyword `var` can also be used, and is more idiomatic when there is just one variable, but it is actually synonymous with `vars`.

Warning: The final blank and period are required for variable declarations.

The syntax for an ordinary equation in OBJ3 is

`eq` ⟨*Term*⟩ `=` ⟨*Term*⟩ `.`

where the two ⟨*Term*⟩s must be well formed OBJ3 terms in the operators and variables available in the current context, and must have a common (super-) sort.

Warning: Equations must be terminated by a blank followed by a period[7]. OBJ will think a "loose" period within an equation marks the end of the equation, and then will probably generate a parse error and other chaos. However, it is easy to avoid this by placing parentheses around an expression that contains the offending period. For similar reasons, any use of `=` in the left side must be enclosed in parentheses.

Warning: All variables in an equation must have been previously declared. For equations appearing in objects[8], each variable that occurs in the right side must also occur in the left side, and the left side must not be a single variable. All these conditions are checked, and warnings are issued if they fail.

Warning: Correctness of OBJ3's operational semantics normally requires that the least sort of the left side of each equation is greater than or equal to that of its right side. If this condition is not satisfied by an equation, then OBJ3 will add retracts in parsing the right side, but not the left side. Sections 2.3.3 and 2.3.4 below discuss this in more detail and also introduce a refinement.

There is a shorthand notation for giving a name to a ground term. The syntax is

[7] The blank can be omitted if the last character is "special", i.e., a parenthesis or bracket.
[8] This requirement is not made for theories (see Section 4.1).

```
let ⟨Sym⟩ = ⟨Term⟩ .
let ⟨Sym⟩ : ⟨Sort⟩ = ⟨Term⟩ .
```

The name used must be single simple symbol (such as "`x`", "`@2`", or "`ThePoint`"). The second form above is equivalent to

```
op ⟨Sym⟩ : -> ⟨Sort⟩ .
eq ⟨Sym⟩ = ⟨Term⟩ .
```

In the first form, the sort of the top operator is taken to be the sort of the term (as discovered by the parser). For example,

```
let t = 1 0 1 .
```

defines `t` to be the list `1 0 1`. A variation of `let` that is suitable for use when applying equations by hand is discussed in Section 5.1.

Warning: The symbol given to a `let` declaration must be a single token.

Warning: When a symbol defined by a `let` is used in a term being reduced, it is replaced by its original definition. Hence, if the symbol is used more than once, the definition will also be reduced more than once.

OBJ also has **conditional equations**, which have the syntax

```
cq ⟨Term⟩ = ⟨Term⟩ if ⟨Term⟩ .
```

where the first two ⟨*Term*⟩s must have a common sort, and the third ⟨*Term*⟩, which is called the **condition**, must have sort `Bool`, a predefined sort that is automatically provided in every module. A conditional rewrite rule can be thought of as a "conditional pattern-driven demon" that awakens when the pattern in its left side is matched and when its condition evaluates to `true`, using values for variables determined by the match. The keyword `ceq` is synonymous with `cq`.

Warning: For conditional equations appearing in objects, all variables that occur in the condition must also occur in the left side, otherwise a warning is issued. Also, any occurrence of `if` in the right side must be enclosed in parentheses, or else OBJ will assume that what follows it is the condition.

The command

```
show rules [⟨ModExp⟩] .
```

will show the rules for the named module (see Section 4.5 for details about module expressions) or for the current module if none is named. Each rule is associated with a positive integer by the system[9]. The rules of certain predefined modules, in particular `BOOL`, will not be displayed unless the command

```
set all rules on .
```

has been executed, in which case all rules from all imported modules are shown. Of course,

```
set all rules off .
```

restores the default. The command

```
show all rules .
```

shows rules as if both the `verbose` (see Section 3) mode and the `all rules` were set `on`.

A label can be given to a rule by using the syntax

```
[⟨LabelList⟩] ⟨Rule⟩
```

where ⟨*LabelList*⟩ is a comma or blank separated list of identifiers, which must not contain a "`.`" or begin with a digit. For example,

```
[sum0,id+] eq M + 0 = M .
```

[9] This is useful for specifying rules in `apply` commands (see Section 5).

(Actually, the label need not immediately precede the rule, and the form `[label]` can be thought of as setting the label for the next rule to be created.) For example,

```
[def1] let x = 1 0 0 .
```

works as expected, i.e., the label "`def1`" is associated with the rule "`x = 100`" that is generated internally by the use of `let`.

Labels are shown when rules are shown. Certain automatically generated rules have automatically generated labels.

The command

`show rule` ⟨*RuleSpec*⟩ `.`

shows the specified rule, where ⟨*RuleSpec*⟩ is defined as follows:

⟨*RuleSpec*⟩ `::=` [-][⟨*ModId*⟩]`.`⟨*RuleId*⟩
⟨*RuleId*⟩ `::=` ⟨*Nat*⟩ | ⟨*Id*⟩

(The syntactic notation used here is explained in Appendix B below.) For example,

```
show rule .def1 .
```

in the context of the module containing the `let` considered above, shows the rule

```
[def1] eq x = 1 0 0
```

and the variant

`show all rule` ⟨*RuleSpec*⟩ `.`

shows a specific rule in `verbose` mode.

Warning: The initial period in `.def1` is required, but square brackets (e.g., `.[def1]`) must not be used.

Warning: It is possible that rules, as displayed by the `show` command, will be renumbered in curious ways when modules are combined.

2.3.1 Operational Semantics is Reduction

We illustrate computation by term rewriting with a simple `LIST-OF-INT` object. (The line `protecting INT` in the example below indicates that the `INT` module, for integers, is imported; module importation is discussed in Section 3.1 below.)

```
obj LIST-OF-INT is
  sort List .
  protecting INT .
  subsort Int < List .
  op __ : Int List -> List .
  op length_ : List -> Int .
  var I : Int .   var L : List .
  eq length I = 1 .
  eq length(I L) = 1 + length L .
endo
```

The subsort declaration "`Int < List`" yields a syntax in which single integers, such as "5", are valid lists. Omitting the parentheses in the last equation above creates a relatively subtle parsing ambiguity, which as an exercise, the reader is invited to discover.

Let us now evaluate some terms over this object. A term ⟨*Term*⟩ to be evaluated is presented with the syntax

`reduce` [`in` ⟨*ModExp*⟩ `:`] ⟨*Term*⟩ `.`

which is evaluated in the context of the module currently in focus, unless the optional "`in` ⟨*ModExp*⟩" is given, in which case it is evaluated in the context of ⟨*ModExp*⟩. Usually the module currently in focus is the last module entered into the system, but this can be changed by using the `select` command, as described in the beginning of Section 3. The keyword `reduce` can be abbreviated to `red`. The term given for reduction may contain variables, in which case a warning is given, but the reduction is carried out with the variables being treated as constants of the declared sorts.

Warning: The period after a term to be reduced is required, and "loose" periods inside the term will confuse the parser and may cause chaos. For example, in

```
reduce 0 . 1 . 0 . nil .
```

OBJ first reduces just "0", and then try to interpret "`1 . 0 . nil .`" as further top level commands; this fails, and produces some further error messages. This can be avoided by enclosing the entire term to be reduced in parentheses. A command like "`red in(1) .`" also fails, because OBJ assumes that the "`in`" introduces a module name. Parentheses can also be used to resolve this ambiguity, as in "`red (in(1)) .`"

A reduce command is executed by matching the given term with the left sides of equations, and then replacing the matched subterm with the corresponding substitution instance of the right side; i.e., evaluation proceeds by applying rewrite rules. For example, the command

```
reduce length(17 -4 329).
```

causes the given term to be evaluated in the module `LIST-OF-INT` if it follows that module, and in this case, the following is printed,

```
==========================================
reduce in LIST-OF-INT : length (17 (-4 329))
rewrites: 5
result NzNat: 3
==========================================
```

as a result of the following sequence of rewrite rule applications,

```
length(17 -4 329) =>
1 + length(-4 329) =>
1 + (1 + length 329) =>
1 + (1 + 1) =>
1 + 2 =>
3
```

which we call a **trace** of the computation. Here, the first step uses the second rule, with the left side `length(I L)` matching `I` to `17` and `L` to `-4 329`. The second step also uses this rule, but now matching `I` to `-4` and `L` to `329`; this match works by regarding the integer `329` as a `List`, because `Int` is a subsort of `List`. The third step simply uses the first rule, and the last steps use the built-in arithmetic of `INT`. Execution proceeds until reaching a term to which no further rules can be applied, called a **normal** (or **reduced**) **form**[10]. The command

```
set trace on .
```

causes a local trace to be printed as a reduction is executed. This displays information describing the application of each rule. Global tracing is produced by the command

```
set trace whole on .
```

which displays the whole term being reduced at each rule application. Similarly, the commands

```
set trace off .
set trace whole off .
```

[10]Most functional programming languages require users to declare **constructors** such that a term is reduced iff it consists entirely of constructors. OBJ3 does not make any use of constructors, and thus achieves greater generality; however, constructor declarations could be used, e.g., for compiler optimization; and they are also used in Kumo to support induction [65].

return OBJ3 to its default state of not printing traces.

The operational semantics for a conditional rewrite rule is as follows: first find a match for the left side; then evaluate the condition, after substituting the bindings determined by the match; if it evaluates to `true`, then do the replacement by the right side, again using the values for the variables determined by the match. Note that evaluating the condition could require non-trivial further rewriting in some cases. This requires OBJ to keep track of bindings, because a term may match a rule in more than one way, and we do not want to keep trying the same match over and over; this bookkeeping can be highly non-trivial for associative/commutative matching (see Section 2.4).

OBJ3 has a built-in (i.e., predefined) polymorphic binary infix `Bool`-valued **equality** operator which specializes as neeeded to any sort `S`; its syntax is

```
op _==_ : S S -> Bool .
```

This operator tests whether or not two ground terms are equal, by reducing the two terms to normal form, and then comparing the normal forms for syntactic identity[11]. For example, `_==_` on `Bool` itself is just `_iff_`. The operator `==` really is equality on a sort provided that the rules for terms of that sort are Church-Rosser, that the rules are terminating with respect to the given evaluation strategy, and that the evaluation strategy is non-lazy (these notions are discussed in Sections 2.3.4 and 2.4.4 below), because these conditions guarantee that normal forms will be reached. The negation `_=/=_` of `_==_` is also available, and so is polymorphic `if_then_else_fi`; these polymorphic operators are all provided by the predefined module `BOOL`, which is automatically imported into each module (unless this default is specifically disabled).

Release 2 of OBJ3 allows variables that are declared in the current context to occur in terms that are presented to the `reduce` and `parse` commands; a warning is issued in the case of reduction. Of course, a parse error will occur if there are variables that have not been declared. For example,

```
reduce length I .       ***> should be: 1
reduce length (I I).    ***> should be: 2
reduce length (I I I).  ***> should be: 3
```

(A parsing error will result from omitting the parentheses, for reasons to be discussed in Section 2.4; "`***>`" indicates that what follows is a comment that should be printed.)

It is possible to perform a number of reductions over the same module in a "reduction loop", with the syntax

⟨*RedLoop*⟩ `::= rl {. |` ⟨*ModId*⟩`} {` ⟨*Term*⟩ `.}... .`

The terms are read, and the reduction results printed, one at a time. If "`.`" is given instead of a ⟨*ModId*⟩, then the current module is used. A synonym for `rl` is `red-loop`.

For example,

```
rl NAT
  5 + 5 .
  3 * 6 + 18 .
  (21 - 8) * 3 . .
```

(the last term will fail to parse, because `-` is not defined in `NAT`).

Sometimes when we want to execute test cases for some code, it may be convenient to use the "`test reduction`" command, which has the syntax

`test reduction [in` ⟨*ModExp*⟩ `:]` ⟨*Term*⟩ `expect:` ⟨*Term*⟩ `.`

and checks whether the result is as expected, and then issues an error message if it isn't. For example,

```
test reduction in NAT : 5 + 5 expect: 10 .
```

But it is often easier to use a comment, such as

```
***> should be: 10
```

or to execute

```
red 5 + 5 == 10 .
```

where `==` is the predefined polymorphic equality function (as described in Section 2.3.1).

[11] If `==` is used for two terms with incompatible sorts, then a parse error occurs.

2.3.2 Denotational Semantics

Whereas an *operational* semantics for a language should show how its computations are done, a *denotational* semantics should give precise mathematical meanings to programs in a way that is as conceptually clear and simple as possible, and that supports proving properties of programs. If a language is rigorously based upon logic, then the already established proof and model theories of the underlying logical system apply *directly* to its programs, and complex formalisms like Scott-Strachey semantics or Hoare logics are not needed. The denotational semantics of OBJ is algebraic, as in the algebraic theory of abstract data types [84, 83, 153, 90], and in particular, the denotation of an OBJ object is an **algebra**, a collection of sets with functions among them[12]. The initial algebra approach [83, 116] takes the unique (up to isomorphism) **initial** algebra as the "standard," or "most representative" model of a set of equations (there may of course be many other models), i.e., as the representation-independent standard of comparison for correctness. It is shown in [11] (see also [116]) that an algebra is initial if and only if it satisfies the following properties:

1. **no junk:** every element can be named using the given constant and operator symbols; and
2. **no confusion:** all ground equations true of the algebra can be proved from the given equations.

For canonical systems (as defined in Section 2.4.4 below), the rewrite rule operational semantics agrees with initial algebra semantics, in the sense that the reduced forms constitute an initial algebra (this result was shown in [46]; see also [116, 151]). Because OBJ3 is based on order sorted algebra, it is important to note that this result easily extends to this context. OSA, and thus OBJ3, provides a completely general programming formalism, in the sense that any partial computable function can be defined[13]. The formalism is especially convenient and natural for non-numerical processing, but in fact, it also handles numerical applications quite felicitously.

2.3.3 Exceptions and Retracts

Exceptions have both inadequate semantic foundations and insufficient flexibility in most programming and specification languages, including functional programming languages. Algebraic specification languages sometimes use partial functions, which are simply undefined under exceptional conditions. Although this can be developed rigorously, as in [98], it is unsatisfactory in practice because it does not allow error messages or error recovery. For some time, we have been exploring rigorous approaches that allow users to define their own exception conditions, error messages, and exception handling. OBJT and OBJ1 used error algebras [43], which sometimes fail to have initial models [131]; however, the current approach based on OSA seems entirely satisfactory to us.

As a simple example, let's consider the natural and rational numbers, with sorts `Nat < Rat`. If `_+_` is only defined for rationals, then `(2 + 2)` is fine because `2` is a natural number and `Nat < Rat`. On the other hand, given the term `(-4 / -2)!`, where `!` is only defined for natural numbers, the parser must consider the subterm `(-4 / -2)` to be a rational, because at parse time it cannot know whether this term will evaluate to a natural number; thus, the term `(-4 / -2)!` does not parse in the conventional sense. However, we can "give it the benefit of the doubt" by having the parser insert a **retract**, which is a special operator symbol (in this case denoted `r:Rat>Nat` and having arity `Rat` and coarity `Nat`) that is removed at run time if the subterm evaluates to a natural, but otherwise remains behind as an informative error message. Thus, the parser turns the term `(-4 / -2)!` into the term `(r:Rat>Nat(-4 / -2))!`, which at runtime becomes first `(r:Rat>Nat(2))!` and then `(2)!`, using the (built-in) **retract equation**

```
r:Rat>Nat(X) = X
```

where `X` is a variable of sort `Nat`. The retract operator symbols are automatically generated by the OBJ3 system, along with the corresponding retract equations. Retracts are inserted, if needed, by the parser, and will not be seen by the user in ordinary examples. However, the user who wants to see retracts can give the command

```
set show retracts on .
```

which causes the OBJ3 system to print them, both when showing equations in modules, and also when showing the results of rewriting. Of course,

[12] We will see later that the denotation of an OBJ theory is a class of algebras, that are not in general isomorphic to one another.

[13] See [6, 116] for similar results about total computable functions.

```
set show retracts off .
```

restores the default mode in which retracts are not shown.

We will see later that retracts can be used to handle rewrite rules that are not sort decreasing (see Section 2.3.4); also, equations with retracts on their left sides are useful for defining "coercions" among various data types, and data with multiple representations, as explained in [73] and illustrated in [50] and [61].

Turning to theory for a moment, retracts are new operators $r_{s,s'} : s \to s'$, one for each pair s, s' of sorts in the same connected component; these give an extension of the original signature provided by the user. For each $r_{s,s'}$, a retract equation $r_{s,s'}(x) = x$ is also added, where x is a variable of sort s. Then (a slight extension of) the "conservative extension" theorem proved in [76] shows that under some mild assumptions, adding these operators and equations does not create any confusion among terms that do not involve retracts. The OBJ3 implementation uses the notation `r:`s`>`s' for the operator $r_{s,s'} : s \to s'$. Details of the mathematical and operational semantics of retracts, using order sorted algebra and order sorted term rewriting, are given in [62] and [76]; see [56] for recent applications of retracts to theorem proving for partial functions.

Warning: Release 2 of OBJ3 does not allow qualified sort names in retracts within terms provided by the user.

Now some code that illustrates retracts. Stacks are a well known benchmark in this area, because the example is simple, but raises the interesting problem of what a term like `top(empty)` actually means, and indeed, whether it has any meaning. The OBJ3 code given below not only handles the exceptions in a natural way, but also seems about as simple as one could hope. The approach is to define a subsort `NeStack` of non-empty stacks, and then say that `top` *is only defined* on this subsort.

```
obj STACK-OF-NAT is sorts Stack NeStack .
  subsort NeStack < Stack .
  protecting NAT .
  op empty : -> Stack .
  op push : Nat Stack -> NeStack .
  op top_ : NeStack -> Nat .
  op pop_ : NeStack -> Stack .
  var X : Nat .   var S : Stack .
  eq top push(X,S) = X .
  eq pop push(X,S) = S .
endo
```

Then evaluating

```
reduce top push(1,empty) .
```

yields the natural number `1`, while

```
reduce pop push(1,empty) .
```

yields `empty`, and

```
reduce top empty .
```

yields

```
result Nat: top r:Stack>NeStack(empty)
```

with `empty` retracted to the sort `NeStack`. Similarly,

```
reduce top pop empty .
```

yields

```
result Nat: top r:Stack>NeStack(pop r:Stack>NeStack(empty))
```

If the `show retracts` mode is `on`, when OBJ3 shows the term to be reduced in the above example, then it will contain retracts; in fact, it will show the same term given above as output, because no reduction is possible.

An alternative approach to exceptions involves introducing supersorts that contain specific error messages for exceptional conditions, as in the following:

```
obj STACK-OF-NAT is
  sorts Stack Stack? Nat? .
  subsort Stack < Stack? .
  protecting NAT .
  subsort Nat < Nat? .
  op empty : -> Stack .
  op push : Nat Stack -> Stack .
  op push : Nat Stack? -> Stack? .
  op top_ : Stack -> Nat? .
  op pop_ : Stack -> Stack? .
  op topless : -> Nat? .
  op underflow : -> Stack? .
  var X : Nat .   var S : Stack .
  eq top push(X,S) = X .
  eq pop push(X,S) = S .
  eq top empty = topless .
  eq pop empty = underflow .
endo
```

Here are some sample reductions for this code:

```
reduce top push(1,empty) .  ***> should be: 1
reduce pop push(1,empty) .  ***> should be: empty
reduce top empty .          ***> should be: topless
reduce pop empty .          ***> should be: underflow
reduce top pop empty .      ***> should be: top r:Stack?>Stack(underflow)
```

Sometimes we may want a certain operator, or a certain term, to have a lower sort than it otherwise would. **Sort constraints** [62, 73] are declarations of this kind. Release 2 of OBJ3 has syntax for a kind of sort constraint that restricts the domain of a multi-argument operator to arguments that satisfy some equational conditions. For example, the code in Section C.8 contains the operator declaration

```
op-as _;_ : Mor Mor -> Mor for M1 ; M2 if d1 M1 == d0 M2 [assoc] .
```

which means that `M1 ; M2` has sort `Mor` if `d1 M1 == d0 M2`, and otherwise has sort `Mor?`, which is an automatically provided error supersort of `Mor`; the attribute `[assoc]` says that `_;_` is associative (the `[assoc]` attribute is discussed in the following subsection).

Warning: Release 2 of OBJ3 only supports the *syntax*, but not the semantics, of sort constraints. In particular, error supersorts (such as `Mor?`) are *not* automatically generated for each user declared sort. However, we hope that some future version of OBJ will fully implement this feature, as it seems to have many interesting applications. In fact, in Release 2 of OBJ3, the semantics of the above `op-as` declaration is equivalent to that of the following ordinary operator declaration:

```
op _;_ : Mor Mor -> Mor [assoc] .
```

A related feature allows defining a subsort by a Boolean expression. For example,

```
sort PosRat .
subsort PosRat < Rat .
var N : Rat .
as PosRat : N if N > 0 .
```

defines `PosRat < Rat` to have as its elements the rationals `N` such that `N > 0`. The syntax is

`as` ⟨*Sort*⟩ `:` ⟨*Term*⟩ `if` ⟨*Term*⟩ `.`

Warning: This feature is not yet implemented, and attempting to use it produces the message

```
Error: general sort constraint not currently handled (ignored)
```

2.3.4 More on the Operational Semantics

This section gives an informal introduction to some delicate aspects of OBJ3's operational semantics; fuller treatments of order sorted rewriting, order sorted equational deduction, and retract equations can be found in [101, 76, 141]. The aim here is to familiarize OBJ3 users with the basic properties that equations should have for term reduction to behave properly. As already mentioned in Section 2.2, OBJ3 assumes that signatures are regular and coherent, and we maintain this assumption throughout this subsection. We first discuss the Church-Rosser property and termination, and then we explain some additional conditions required to handle subsorts.

Often, the order of applying rules does not affect the result, in the sense that whenever a term t_0 is rewritten in two different ways, obtaining terms t_1 and t_2, then there is another term t_3 such that both t_1 and t_2 rewrite to t_3. A rule set with this desirable property is called **Church-Rosser**; OBJ3 assumes that the rules in objects are Church-Rosser. Another desirable property for a rule set is **termination**, in the technical sense that there are *no* infinite sequences of rewrite rule applications. A rule set that is terminating (in this sense) can be checked for the Church-Rosser property by the Knuth-Bendix algorithm [104]; a rule set that is both terminating and Church-Rosser is called **canonical**. Although we cannot assume that all rule sets are terminating, rules that define total computable operators over total computable sets can always be chosen to be both Church-Rosser and terminating [6]; this includes the typical case of abstract data types. However, further functions defined over these structures can fail to have terminating rule sets, for example, if they implement procedures for problems that are only semi-decidable, such as full first-order theorem proving, higher-order unification, or combinator reduction[14]. The Knuth-Bendix algorithm extends to a completion procedure that may produce a canonical rule set from one that is terminating. Note that an order sorted version of Knuth-Bendix is needed for OBJ3 [141]. Huet and Oppen give a nice survey of rewrite rule theory which develops some connections with general algebra [93]; Klop [102, 103] and Dershowitz and Jouannaud [24] have also written useful surveys of this area, that are more up to date. OSA foundations for the issues discussed above may be found in [117, 62, 101, 141]. Chapter 5 of [59] gives a recent algebra-oriented survey of term rewriting, and [4] is a recent elementary textbook on this subject.

We have run many thousands of reductions on many hundreds of examples, often in dozens of variations, and we have hardly ever encountered problems with canonicity. We believe that OBJ users almost always write equations for abstract data types that are canonical, because they tend to think of equations as programs, and therefore they write primitive recursive definitions for operators. A practical implication of this is that tests for canonicity are not of critical importance. This is fortunate, because the problem is undecidable.

In summary, the intuition of the Church-Rosser property is that when it holds, reduction can be seen as evaluating a functional expression to a unique result that does not depend on the order of evaluation. The termination property ensures that this result always exists.

We now discuss some issues concerning subsorts in rewriting. Intuitively, the more we advance in evaluating a functional expression, the more information we should have about its result. This should also apply to information about the *sort* of the result, and the smaller the sort that we can associate to a data element, the more information we have about that data element. For example, by syntactic analysis we can only associate to the expression `(7 + (- 3))/ 2` the sort `Rat`, but after evaluation we know that its result has sort `NzNat`.

This suggests that rewrite rules should be **sort decreasing**, i.e., that if a term t can be rewritten to a term t', then the least sort of t' should always be less than or equal to the least sort of t. This is very often the case in the many examples that we have studied; however, there are some quite reasonable rewrite rules that can violate this requirement *temporarily*. For example, in the number hierarchy described in Appendix C.7, the rule

```
eq | C |^2 = C * (C #) .
```

which defines the square modulus of a complex number with rational coordinates as the product of the number by its conjugate, has a left side with sort `Rat` and a right side with sort `Cpx`. In the end, this will (in the appropriate sense clarified below) not matter because the requirement of rewriting down is only violated temporarily, and the reduced expression is always a rational number.

But in general, the careless treatment of rules that are not sort decreasing could result in unsound deductions. For example, consider the object

[14] In order not to add new values to the underlying abstract data type, the value sorts of such potentially non-terminating operators should be error supersorts; then retracts will be added when they are used as ordinary values in terms.

```
obj PROBLEMS is
  sorts A B .
  subsorts A < B .
  op a : -> A .
  op b : -> B .
  ops f g : A -> A .
  var X : A .
  eq f(X) = g(X) .
  eq a = b .
endo
```

and suppose that we want to reduce the term `f(a)`. By applying the first equation, we can reduce it to `g(a)`, and by applying the second equation, we can reduce it to `g(b)`. The problem here is that `g(b)` is not a well-formed term! In fact, the second step of deduction is not allowed under the rules of order sorted equational deduction [76], and is unsound in this precise sense. The problem is that while the deduction `a = b` is sound in itself, it becomes unsound in the context of the enclosing function symbol `g`.

Retracts allow a correct and sound treatment of rules like that for the square modulus of a complex number discussed above. In this treatment, rewriting works even if the rules are not sort decreasing, provided they are "reasonable" in a sense that is made precise below. Moreover, if the rules are not reasonable, then the retracts will help to detect flaws in the specification. The idea is as follows: suppose that a term t of least sort s can be rewritten at the top by applying a rule $u = v$ to yield a term t', i.e., suppose there is a substitution θ (respecting the sorts of the variables) such that $t = \theta(u)$ and $t' = \theta(v)$. Next, suppose that the least sort s' of t' is not less than or equal to s. This could be a problem if our term t were enclosed in a subterm that made its replacement by t' ill-formed. A way to guarantee that this never happens, regardless of the embedding context, is to check the least sorts of t and t', and whenever the sort of t is not greater than or equal to the sort of t', to replace t not by t' but by the term $\mathtt{r:}s'\mathtt{>}s(t')$. We then call the rewrite from t to $\mathtt{r:}s'\mathtt{>}s(t')$ a **safe rewrite** using the rule $u = v$. Rewrites in which retract symbols are eliminated by applying the built-in retract rules (see Section 2.3.3), and also rewrites with $s \geq s'$, where no retract is needed, are also considered safe.

The key point about safe rewrites is that they are *sound*, i.e., they are correct logical deductions. First, as discussed in Section 2.3.3, adding retracts and retract rules to the original specification is conservative, in the sense that no new equalities between terms without retracts can be derived after the addition of retracts and retract rules. The soundness of safe rewriting then follows from the observation that the rewrite from t to $\mathtt{r:}s'\mathtt{>}s(t')$ using the rule $u = v$ is a combination of two sound steps of deduction, namely, we can first derive $\mathtt{r:}s'\mathtt{>}s(t) = \mathtt{r:}s'\mathtt{>}s(t')$ using $u = v$, and then derive $t = \mathtt{r:}s'\mathtt{>}s(t')$ by applying to the term on the left the retract rule $\mathtt{r:}s'\mathtt{>}s(x) = x$ with x a variable of sort s.

Notice that if the original rules are sort decreasing, then safe rewrites and ordinary rewrites coincide, in the sense that there is never any need to introduce retracts on the right side after applying a rule. Safe rewriting allows us to broaden the class of rules that OBJ3 can handle properly, to include all intuitively "reasonable" rules. Without safe rewriting, we would have to require that all rules are sort decreasing. But with safe rewriting it is enough to require that if a ground term t without retracts can be safely rewritten to a term t' to which no rules can be applied, then

(1) t' has no retracts; and

(2) any sequence of safe rewrites from t to t'' can be continued by a sequence of safe rewrites from t'' to t'.

We call rules satisfying conditions (1) and (2) **reasonable** rules. Notice that, by the definition of safe rewriting, an irreducible term t', as well as any intermediate term, will have a least sort less than or equal to the least sort of the original term t. Notice also that any rules that are Church-Rosser and sort decreasing are obviously reasonable. The rules for the complex number example are reasonable (and also terminating). The rules in the `PROBLEMS` example are unreasonable, and the user will get evidence of this by performing reductions. For example, the reduction of `a` will yield the result `r:B>A(b)`, and the reduction of `f(a)` will yield the result `g(r:B>A(b))`, both of which violate condition (1).

One last word of caution. Failure to be reasonable may not be apparent from a cursory inspection of the rules. For example, consider the object

```
obj MORE-PROBLEMS is
  sorts A B C .
  subsorts A < B < C .
```

```
  op f : C -> C .
  ops f h : A -> A .
  op g : B -> B .
  op a : -> A .
  var X : B .
  eq f(X) = g(X) .
endo
```

Here, `f(X)` has a sort `C` that is greater than `B`, the sort of `g(X)`. However, this specification is unreasonable. For example, the term `h(f(a))` reduces to the term `h(r:B>A(g(a)))`, violating condition (1). The problem is that the rule

```
eq f(Y) = g(Y) .
```

obtained from the original rule by "specializing" the variable `X` of sort `B` to a variable `Y` of sort `A` violates the sort decreasing property. Therefore, not just the original rules, but also all of their "specializations" to rules with variables having smaller sorts may have to be considered; see [101].

2.4 Attributes

It is natural and convenient to consider certain properties of an operator as **attributes** that are declared at the same time as its syntax. These properties include axioms like associativity, commutativity, and identity that have both syntactic and semantic consequences, as well as others that affect order of evaluation, parsing, etc. In OBJ3, such attributes are given in square brackets after the syntax declaration. You can see what attributes an operator actually has with the `show` command, which has the following syntax,

`show op` ⟨*OpRef*⟩ `.`

where ⟨*OpRef*⟩ describes the operator (see Appendix B for its details). The operator will be described in the context of the module currently in focus.

2.4.1 Associativity and Commutativity

Let us first consider associativity. For example,

```
op _or_ : Bool Bool -> Bool [assoc] .
```

indicates that `or` is an associative binary infix operator on Boolean values. This implies that the parser does not require full parenthesisation. For example, we can write `(true or false or true)` instead of `(true or (false or true))`; moreover, the term printer will omit unnecessary parentheses. Of course, the `assoc` attribute also gives the semantic effect of an associativity axiom, which is implemented by associative rewriting and associative extensions, as described below.

Warning: The `assoc` attribute is only meaningful for a binary operator with arity `A B` and value sort `C` when `C < A` and `C < B`; however, retracts might be inserted if either `A < C` or `B < C`.

Binary infix operators can be declared commutative with the attribute `comm`, which is semantically a commutativity axiom, implemented by commutative rewriting (as described below). Note that a binary operator can be given both commutative and associative attributes.

Warning: The commutative attribute is only meaningful when the two sorts in the arity have a common supersort; also, some retracts may be added if these two sorts are unequal.

Warning: Because associative/commutative matching is an NP-complete problem, a *uniformly fast* implementation is impossible.

The present implementation, based on work of Lincoln [108] extended to OSA along the lines of [62, 101], is reasonably efficient, but cannot be expected to run quickly for really large problems; see also [96].

2.4.2 Identity and Idempotence

An identity attribute can be declared for a binary operator. For example, in

```
op _or_ : Bool Bool -> Bool [assoc id: false] .
```

the attribute `id: false` gives the effects of the identity equations `(B or false = B)` and `(false or B = B)`. Identity attributes can be *ground terms* and not just constants.

Warning: All the operators occurring in the value term of an identity attribute must have been previously declared.

If it only makes sense to have a left or a right identity, then that is all that is generated. For example, in

```
op nil : -> List .
op __ : Int List -> NeList [id: nil] .
```

only a right identity equation is added. A left identity equation is added if the sort of the identity is a subsort of the left arity sort, and a right identity equation is added if the sort of the identity is a subsort of the right arity sort.

Warning: OBJ3 implements rewriting modulo identity by a combination of direct matching modulo identity, and a partial completion process that may generate further equations. Matching modulo identity very often leads to problems with termination and efficiency, as discussed in Section 3.1.1 below.

The attribute `idr:` introduces only the identity equations themselves, without invoking any completion process. This can be convenient for avoiding the termination problems associated with the `id:` attribute.

Warning: Associative, commutative and identity attributes are inherited downward, from an overloaded operator to all operators having the same form and lower rank.

Operators can also be declared idempotent, by using the attribute `idem`; this is implemented simply by adding the idempotent equation.

Warning: The effect of rewriting modulo idempotence is neither attempted nor achieved.

It is possible to give *any* operator symbol any of the attributes `assoc`, `comm`, `id:`, `idr:` and/or `idem`; warnings are issued when the attributes do not make sense.

Let us now consider a more sophisticated integer list object with associative and identity attributes,

```
obj LIST-OF-INT1 is
  sorts List NeList .
  protecting INT .
  subsorts Int < NeList < List .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil] .
  op __ : NeList List -> NeList [assoc id: nil] .
  op head_ : NeList -> Int .
  op tail_ : NeList -> List .
  var I : Int .   var L : List .
  eq head(I L) = I .
  eq tail(I L) = L .
endo
```

and some test cases using this object. For example,

```
reduce 0 nil 1 nil 3 .
```

is carried out in `LIST-OF-INT1` by applications of the identity equation modulo associativity, as follows,

```
0 nil 1 nil 3 =>
0 1 nil 3 =>
0 1 3
```

and it prints

```
result NeList: 0 1 3
```

Similarly, we may consider

```
reduce head(0 1 3) .         ***> should be: 0
reduce tail(0 1 3) .         ***> should be: 1 3
reduce tail(nil 0 1 nil 3) . ***> should be: 1 3
```

2.4.3 Precedence and Gathering

Ambiguity in the parsing of terms can be reduced by using precedence and gathering. The **precedence** of an operator is a number (in the range 0 to 127), where a lower value indicates "tighter binding" in the sense that is discussed below. For example, the predefined object `INT` contains the declarations

```
op _+_ : Int Int -> Int [assoc comm id: 0 prec 33] .
op _*_ : Int Int -> Int [assoc comm id: 1 prec 31] .
```

The precedence of a *term* is the precedence of its top operator, unless it is enclosed in parentheses or qualified, in which case it has precedence `0`. Ordinarily, the arguments of an operator must have precedence less than or equal to its precedence. Therefore, `(1 + 2 * 3)` is not parsed as `((1 + 2) * 3)`, but instead is parsed as `(1 + (2 * 3))` under the above declarations, because the precedence of the arguments to `*` must be less than or equal to `31`. Intuitively, we can think of the "tighter binding" indicated by lower precedence as the strength with which an operator "pulls" on its arguments; in this example, the constant `2` has been pulled on more strongly by `*` than by `+`.

The default precedence for an operator with standard (i.e., prefix-with-parentheses) form is `0`. If an operator pattern begins and ends with something other than an underbar, then its precedence also defaults to `0`. Unary prefix operators have default precedence `15`. In all other cases, the default precedence is `41`.

This default behavior can be modified. The **gathering pattern** of an operator is a sequence of elements `e`, `E`, or `&` (one element for each argument position) that restricts the precedences of terms that are allowed as arguments: `e` indicates that the corresponding argument must have strictly lower precedence, `E` allows equal or lower precedence, and `&` allows any precedence. For example, parentheses could be described as having precedence `0` and gathering pattern `(&)`; also, the gathering pattern `(E e)` forces left association for a binary operator.

An interesting example that needs this gathering pattern is combinatory algebra, the code for which (see Appendix C.6) includes the following declaration:

```
op __ : T T -> T [gather (E e)] .
```

For example, under this declaration, `a b c` will be parsed as `(a b) c`, rather than as `a (b c)`.

The default gathering pattern for an operator with standard form is all `&`s. If an underbar for an argument position is not adjacent to another underbar, then the default gathering value for that position is `&`. In all other cases, the default gathering value is `E`.

Warning: The current OBJ3 parser sometimes "jumps to conclusions" based on precedence and gathering information, and then simply fails if its initial assumption is wrong. This means that sometimes a term that really does have a unique parse of least sort may fail to parse. Although the defaults are surprisingly effective, sometimes it is necessary to explicitly give carefully chosen precedence and gathering attributes, and/or to insert some parentheses into terms, in order to get the parsing behavior that you want.

2.4.4 Order of Evaluation

In general, a large tree will have many different sites where rewrite rules might apply, and the choice of which rules to try at which sites can strongly affect efficiency, and can also affect termination. Most modern functional programming languages have a uniform lazy (i.e., top-down, or outermost, or call-by-name) semantics. But because raw lazy evaluation is slow, lazy evaluation enthusiasts have built clever compilers that figure out when an "eager" (i.e., bottom-up or call-by-value) evaluation can be used with exactly the same result; this is called "strictness analysis" [123, 95]. OBJ3 is more flexible, because each operator can have its own evaluation strategy. Moreover, the OBJ3 programmer gets this flexibility with minimum effort, because OBJ3 determines a default strategy if none is explicitly given. This default strategy is computed very quickly, because only a very simple form of strictness analysis is done, and it is surprisingly effective, though of course it does not fit all possible needs.

Syntactically, an **E-strategy** (E for "evaluation") is a sequence of integers in parentheses, given as an operator attribute following the keyword `strat`. For example, OBJ's built-in conditional operator has the following E-strategy,

```
op if_then_else_fi : Bool Int Int -> Int [strat (1 0)] .
```

which says to evaluate the first argument until it is reduced, and then apply rules at the top (indicated by "`0`"). Similarly,

```
op _+_ : Int Int -> Int [strat (1 2 0)] .
```

indicates that `_+_` on `Int` has strategy `(1 2 0)`, which says to evaluate both arguments before attempting to add them.

The default E-strategy for a given operator is determined from its equations by requiring that all argument places that contain a non-variable term in some rule are evaluated before equations are applied at the top. If an operator with a user-supplied strategy has a tail recursive rule (in the weak sense that the top operator occurs in its right side), then OBJ3 may apply an optimization that repeatedly applies that rule, and thus violates the strategy. In those rare cases where it is desirable to prevent this optimization from being applied, you can just give an explicit E-strategy that does not have an initial `0`.

There are actually two ways to get lazy evaluation. The simplest approach is to omit a given argument number from the strategy; then that argument is not evaluated unless some rewrite exposes it from underneath the given operator. For example, this approach to "lazy cons" gives

```
op cons : Sexp Sexp -> Sexp [strat (0)] .
```

The second approach involves giving a negative number `-j` in a strategy, which indicates that the $\mathtt{j}^{th}$ argument is to be evaluated "on demand," where a "demand" is an attempt to match a pattern to the term that occurs in the $\mathtt{j}^{th}$ argument position. This approach to lazy cons is expressed as

```
op cons : Sexp Sexp -> Sexp [strat (-1 -2)] .
```

Then a `reduce` command at the top level of OBJ3 is interpreted as a top-level demand that may force the evaluation of certain arguments. This second approach cannot be applied to operators with an associative or commutative attribute. Appendix C.5 gives a further example of lazy evaluation, using the Sieve of Erasthothenes to find all prime numbers.

A strategy is called **non-lazy** if it requires that all arguments of the operator are reduced in some order, and either the operator has no rules, or the strategy ends with a final "`0`". In general, in order for all subterms of a reduction result to be fully reduced, it is necessary that all evaluation strategies be non-lazy. The default strategies computed by the system are non-lazy.

2.4.5 Memoization

Giving an operator the `memo` attribute causes the results of evaluating a term headed by this operator to be saved; thus the work of reduction is not repeated if that term appears again [118]. In OBJ3, the user can give any operators that he wishes the `memo` attribute, and this is implemented efficiently by using hash tables. More precisely, given a memoized operator symbol `f` and given a term `f(t1,...,tn)` to be reduced (possibly as part of some larger term), a table entry for `f(t1,...,tn)` giving its fully reduced value is added to the memo table. Moreover, entries giving this fully reduced value are also added for each term `f(r1,...,rn)` that, according to the evaluation strategy for `f`, could arise while reducing `f(t1,...,tn)` just before a rule for `f` is applied at the top; this is necessary because at that moment the function symbol `f` could disappear. In some cases, memoizing these intermediate reductions is more valuable than memoizing just the original expression.

For example, suppose that `f` has the strategy `(2 3 0 1 0)`, let `r` be the reduced form of the term `f(t1,t2,t3,t4)`, and let `r`i be the reduced form of `t`i for $i = 1, 2, 3$. Then the memo table will contain the following pairs:

```
(f(t1,t2,t3,t4),r)
(f(t1,r2,r3,t4),r)
(f(r1,r2,r3,t4),r)
```

Memoization gives the effect of structure sharing for common subterms, and this can greatly reduce term storage requirements in some problems. Whether or not the memo tables are re-initialized before each reduction can be controlled with the top level commands

```
set clear memo on .
set clear memo off .
```

The default is that the tables are *not* reinitialized. However, they can be reinitialized at any time with the command

```
do clear memo .
```

Each of these commands must be terminated with a blank followed by a period. Of course, none of this has any effect on the *result* of a reduction, but only on its speed. A possible exception to this is the case where the definitions of operators appearing in the memo table have been altered. (When rules are added to an `open` module, previous computations may become obsolete. Therefore, you may need to explicitly give the command "`do clear memo .`"; see Section 3.2.) Memoization is an area where term rewriting based systems seem to have an advantage over unification based systems like Prolog.

2.4.6 Propositional Calculus Example

This subsection gives a decision procedure for a theory of real interest, the propositional calculus. The procedure is due to Hsiang [92], and makes crucial use of associative/commutative rewriting. The OBJ3 code for the object `PROPC` below evolved from OBJ1 code originally written by David Plaisted [77]. It reduces tautologuous propositional formulae, in the usual connectives (`and`, `or`, `implies`, `not`, `xor` (exclusive or) and `iff`) to the constant `true`, and reduces all other formulae to a canonical form (modulo the commutative and associative axioms) in the connectives `xor`, `and`, `true` and `false`. The `TRUTH` object used here contains just `true` and `false` (plus the basic `true, false`-valued operators `_==_, _=/=_` and `if_then_else_fi`), while `QID` provides identifiers that begin with an apostrophe, e.g., `'a`. The module import modes `extending` and `protecting` are discussed in Section 3.1 below. The rules in this object have been shown by Hsiang [92] to be Church-Rosser and terminating modulo the commutative and associative axioms.

```
obj PROPC is
  sort Prop .
  extending TRUTH .
  protecting QID .
  subsorts Id Bool < Prop .
    *** constructors ***
  op _and_ : Prop Prop -> Prop [assoc comm idem idr: true prec 2] .
  op _xor_ : Prop Prop -> Prop [assoc comm idr: false prec 3] .
  vars p q r : Prop .
  eq p and false = false .
  eq p xor p = false .
  eq p and (q xor r) = (p and q) xor (p and r) .
    *** derived operators ***
  op _or_ : Prop Prop -> Prop [assoc prec 7] .
  op not_ : Prop -> Prop [prec 1] .
  op _implies_ : Prop Prop -> Prop [prec 9] .
  op _iff_ : Prop Prop -> Prop [assoc prec 11] .
  eq p or q = (p and q) xor p xor q .
  eq not p = p xor true .
  eq p implies q = (p and q) xor p xor true .
  eq p iff q = p xor q xor true .
endo
```

Now some sample reductions in the context of this object:

```
reduce 'a implies 'b iff not 'b implies not 'a .  ***> should be: true
reduce not('a or 'b) iff not 'a and not 'b .      ***> should be: true
reduce 'c or 'c and 'd iff 'c .                   ***> should be: true
```

```
reduce 'a iff not 'b .                          ***> should be: 'a xor 'b
reduce 'a and 'b xor 'c xor 'b and 'a .         ***> should be: 'c
reduce 'a iff 'a iff 'a iff 'a .                ***> should be: true
reduce 'a implies 'b and 'c iff ('a implies 'b) and ('a implies 'c) .
                                                ***> should be: true
```

Thus, the first three and last two expressions are tautologies, while the fourth is true if and only if exactly one of `'a` and `'b` is true, and the fifth is true iff `'c` is true. Note that `'a`, `'b`, `'c` are **propositional variables** in the sense that anything of sort `Prop` can be substituted for them while still preserving truth; in particular, `true` and `false` can always be substituted. Of course, deciding tautologies in the propositional calculus is an NP-complete problem, so we cannot expect this code to run very fast for large problems.

This example illustrates a striking advantage of using a logical language: every computation is a proof, and interesting theorems can be proved by applying the right programs to the right data. Even if the given equations do not define a decision procedure for a given theory, so long as they are all correct with respect to this theory, then the results of reduction will be correct. For this purpose, we don't even need the Church-Rosser property. For example, even if we didn't know that `PROPC` was canonical, we could still be certain that any term that reduces to `true` is a tautology. Thus, OBJ code can be used for theorem proving, as illustrated by the examples in [51] and [59], a sample from which is given in Section 4.8 below. Some more elaborate theorem proving examples are given in Appendix C.4, and much more detail is available in [66, 59].

3 Module Hierarchies

Conceptual clarity and ease of understanding are greatly facilitated by breaking a program into modules, each of which is mind-sized and has a natural purpose. This in turn greatly facilitates both debugging and reusability. When there are many modules, it is helpful to make the hierarchical structure of module dependency explicit, so that whenever one module uses sorts or operators declared in another, the other is explicitly imported to the first, and is also defined earlier in the program text. A program developed in this way has the abstract structure of an *acyclic graph* of abstract modules[15]. We will use the word **context** to describe such a graph of modules, later extending it to include views, as discussed in Section 4.3.

More exactly now, a directed edge in an acyclic graph of modules indicates that the higher (target) module **imports** the lower (source) module, and the **context** of a given module is the subgraph of other modules upon which it depends, i.e., the subgraph of which it is the top. Parameterized modules can also occur in such a hierarchy, and are treated in essentially the same way as unparameterized modules; they may also have instantiations, and these are considered to be distinct from the parameterized module itself.

In addition to representing program structure in a clear and convenient way, the module hierarchy can have some more specific applications, such as maintaining multiple mutually inconsistent structures as subhierarchies, which could be useful for keeping available more than one way to do the same or related things, for example, in a family of partially overlapping system designs; that is, the module hierarchy can be used for configuration management. It can also be used to keep information from different sources in different places, and to maintain multiple inconsistent worlds, which would be useful in Artificial Intelligence applications exploring the consequences of various mutually inconsistent assumptions, where there may also be some shared assumptions. Hierarchical structure could also be used to reflect access properties of a physically distributed database, as suggested in [48].

The command

```
show modules .
```

shows a list of all modules in the current OBJ working context. If a module with an atomic name has been redefined, then it may appear more than once in the output from this command.

One can save the current context with the command

`do save` ⟨*ChString*⟩ `.`

where ⟨*ChString*⟩ is any character string, possibly containing blanks; you can then return to a previously named context with the command

[15] Such a hierarchy differs from what is sometimes called a "Dijkstra-Parnas" hierarchy, because lower level modules do not *implement* higher level (less abstract) modules, but rather, lower level modules are *included in* higher level modules; both kinds of hierarchy are supported in the LILEANNA system [85].

```
do restore ⟨ChString⟩ .
```

Warning: The commands `save` and `restore` do not work correctly in an OBJ3 system that has been constructed using Kyoto Common Lisp; however, they do work correctly in a system built using (for example) Lucid Common Lisp.

Warning: The command `openr` (as described in Section 3.2) can retroactively change a saved context.

The initial OBJ context, which has exactly what the standard prelude provides, can be restored at any time with the command

```
do restore init .
```

Simply reading in the standard prelude again will restore the modules in the standard prelude to their original state, but will not delete any modules that have been subsequently added.

OBJ3 has a notion of "the module currently in focus", and the module name `THE-LAST-MODULE` evaluates to that module; ordinarily, this is the last module mentioned to the system, but it can be changed to any desired module by using the command

```
select [⟨ModExp⟩] .
```

(The old form `show select` ⟨*ModExp*⟩ still works.)

The following synonymous commands

```
show [⟨ModExp⟩] .
show mod [⟨ModExp⟩] .
```

display the structure of the given module, or of the current module if no ⟨*ModExp*⟩ is given.

Warning: When a module is displayed, some details may omitted, and other details may be shown that the user did not input, for example, some declarations that properly belong to submodules.

The command

```
show all [⟨ModExp⟩] .
```

shows a module in more detail than the default form. The commands

```
set verbose on .
set verbose off .
```

control whether modules are displayed in detailed form by default. (It also controls whether a trace of the id processing is displayed; this is discussed later.)

OBJ3 automatically generates abbreviated names for modules and module expressions; they can be very useful, because sometimes the "official" name of a module can be very long indeed. These abbreviations have the form "`MOD`⟨*Nat*⟩" where ⟨*Nat*⟩ is a natural number, and can serve as names for modules in many top-level commands. One can see the abbreviation for a module's name with the command

```
show abbrev [⟨ModExp⟩] .
```

For example,

```
show abbrev PROPC .
```

gives the abbreviation for `PROPC`, which at the current point in processing the OBJ3 code in this paper is `MOD40`. Names of the form `MOD`⟨*Nat*⟩ can also be used in `show` commands and many other contexts. These names are considered abbreviations, rather than a formal part of the syntax of OBJ3.

3.1 Importing Modules

OBJ3 has four **modes** for importing modules, with the syntax

⟨*ImportKw*⟩ ⟨*ModExp*⟩ .

where ⟨*ImportKw*⟩ is one of `protecting`, `extending`, `including`, or `using`, and ⟨*ModExp*⟩ is a module expression, such as `INT`; the abbreviations `pr`, `ex`, `inc`, and `us` can be used for the corresponding mode keywords.

By convention, if a module `M` imports a module `M'` that imports a module `M''`, then `M''` is also imported into `M`; that is, "imports" is a *transitive* relation. A given module `M'` can only be imported into `M` with a single mode; modules that are multiply imported due to transitivity are ordinarily considered to be "shared."

The meaning of the import modes is related to the initial algebra semantics of objects, in that an importation of module `M'` into `M` is:

1. **protecting** iff `M` adds no new data items of sorts from `M'`, and also identifies no old data items of sorts from `M'` (no junk and no confusion);
2. **extending** iff the equations in `M` identify no old data items of sorts from `M'` (no confusion);
3. **including** or **using** if there are no guarantees at all (see below for the difference between these).

A `protecting` importation has the advantage that it guarantees that no newly generated rules need to be added to the imported module, and also, the E-strategies of imported operators do not need to be recomputed; thus, the code from protecting imported modules can just be shared.

Warning: OBJ3 does not check whether the user's import declarations are correct, because this could require arbitrarily difficult theorem proving that would render the language impractical. However, the consequences of an incorrect import mode declaration can be serious: incomplete reductions in some cases, and inefficient reductions in others.

Warning: If an object `A` has a sort `S`, and an object `B` imports `A` and introduces a new subsort `S'` of `S`, then things may not work as you expect, even if mathematically `A` is protected in `B`. In particular, if `B` introduces a new overloading of an operator of sort `S` from `A` that restricts to `S'`, then the `protecting` declaration may cause failure to generate rules that are needed for matching for some cases, such as rules associated with an identity attribute. Also, retracts may appear on right sides because of equations that appear to be sort increasing (this issue was discussed in Section 2.3.4).

For example, the module

```
obj A is
  sort S .
  op f : S -> S .
  ops a b : -> S .
  vars X : S .
  eq f(X) = b .
endo
```

is certainly protected from the mathematical point of view in the module

```
obj B is
  protecting A .
  sort S' .
  subsorts S' < S .
  op f : S' -> S' .
  op a : -> S' .
endo
```

However, in the context of `B`, the equation `f(X) = b`, which was trivially sort decreasing in `A`, is no longer sort decreasing, and evaluating the term `f(a)` in the context of the module `B` now gives the result `r:S>S'(f(b))`. Thus, introducing new sorts under previous sorts should be avoided in *protecting* importations.

For an `extending` importation, the E-strategies associated to imported operators are recomputed, according to the following rules:

1. if an imported operator has an explicit, user-supplied strategy, then use it;
2. if not, and if there are no new equations, then use the inherited computed strategy; and
3. if there are new equations, then recompute the strategy and use the new one.

`Including` is implemented as incorporation without copying, and in this respect is similar to `protecting`; if a module is included twice in a given module, only one version is created (if it doesn't already exist) and all references are to the same shared instance.

Warning: The `using` mode is implemented by copying the imported module's top-level structure, sharing all of the submodules that it imports. It is required that all copied sorts within a given module have distinct names, and that all copied operators are uniquely identified by their name and rank. This means operators that require qualification will be a problem. Such operators may be mistakenly collapsed into a single operator.

Warning: "`using BOOL`" is not meaningful, because a `using` importation that is not an `extending` importation will identify `true` with `false`, which is not only not useful, but also will interfere with the predefined operators `_==_` and `if_then_else_fi`.

Sometimes it is desirable to copy not only the top-level structure of a module, but also that of some of its submodules, for example, to ensure that the associative or identity completion process is carried out, or that evaluation strategies are recomputed. This can be done using the following syntax,

`using` ⟨*ModExp*⟩ `with` ⟨*ModExp*⟩ {`and` ⟨*ModExp*⟩}`...`

which causes the listed submodules to be copied instead of shared. This feature is illustrated in the unification example in Appendix C.3. Note that all automatically created submodules[16] are automatically copied by `using`, so the multi-level using declaration is not needed for such cases.

The module that introduces a given sort often establishes a convention for naming variables of that sort, and introduces a number of variables for it. The following command makes it easy to reuse variable names, and thus to maintain such conventions. Thus,

`vars-of` ⟨*ModExp*⟩ `.`

introduces all the variables from ⟨*ModExp*⟩; these have the same names and sorts as in ⟨*ModExp*⟩.

Warning: Only the variables declared in ⟨*ModExp*⟩ are introduced, and not variables from modules imported by ⟨*ModExp*⟩, even if their variables had been introduced into ⟨*ModExp*⟩ using `vars-of`.

OBJ3 permits redefining any module, simply by introducing a new module with the old name. A warning is issued, indicating that redefinition has occurred, and then all future mentions of this name refer to the new definition. This can be very useful in theorem proving; for example, you may want to replace a predefined module for numbers that is efficient, by another that is less efficient but more logically complete; Appendix C.4 contains several examples of this.

Warning: Redefining a module does not cause the redefinition of modules that have been previously built from it. For example, if we define `A` to be an enrichment of `INT`, then redefine `INT`, and then look at `A`, it will still involve the old definition of `INT`. The same happens with parameterized modules.

`BOOL` is implicitly `protecting` imported into every module, to ensure that conditional equations can be used, unless an explicit `extending BOOL` declaration is given instead; `TRUTH` can be imported instead of `BOOL` by giving an explicit declaration, as in the `PROPC` example in Section 2.4.6. Usually it is convenient that `BOOL` has been imported, because conditional equations often make use of the operators that are provided in `BOOL`, such as `==`, `and`, or `not`. But sometimes, especially in applications to theorem proving, this can be inconvenient, because it does not provide enough rules to correctly decide all equalities about truth values, even though it does correctly specify the initial algebra of Boolean truth values. The command

```
set include BOOL off .
```

causes not importing `BOOL` to become the default. The original default can be restored with the command

```
set include BOOL on .
```

When `include BOOL` is on, then `BOOL` is included in a module before anything except a `using`, `protecting`, `extending` or `sort` declaration, unless `TRUTH-VALUE`, `TRUTH`, or `BOOL` itself has been included already. (This will affect the determination of the principal sort of a module, as described in Section 4.3.)

[16]This notion is discussed in Section 4.5 below.

3.1.1 Identity Completion and Associative Extensions

Pattern matching for operators with identities is implemented in OBJ3 using a process called **id processing** that consists of a "partial identity completion" process that may generate some new rules, and an "id processing" process that may add some so-called "id conditions" to rules. Identity completion generates instances of a rule by considering "critical pairs" between the rule and the identity equations, in order to give the effect of rewriting modulo identity. For example, consider the first equation from the module `LIST-OF-INT1` in Section 2.4, `head(I L) = I`. This equation has `head I = I` as the special case where `L` = `nil`, and identity completion considers adding it to the rulebase; however, it is not actually added in this case, because the rule is matched in such a way that terms are rewritten as if this rule had been added. Id processing restricts the standard identity completion process to avoid simple cases of non-termination by adding id conditions to rules, so that obviously problematic instances are disallowed, and also by discarding rule instances whose left sides are variables (because their implementation as rules is problematic); in addition, generated rules that are subsumed by other rules are deleted for the sake of efficiency.

Warning: Experience shows that matching modulo identity often results in problems with non-termination. It is safer to use the attribute `idr:` and then add any desired identity completion equations by hand.

Warning: Strategies of operators are not taken into account when testing for non-termination. It is possible that a rule will be considered non-terminating, when this condition is actually avoided because of the evaluation strategies.

For example, in the object `PROPC` of Section 2.4.6, if you replace the attributes `idr:` by `id:`, then identity completion will substitute `q` = `false` into the distributive law,

```
eq p and (q xor r) = (p and q) xor (p and r) .
```

and (in effect) add the new equation

```
eq p and r = (p and false) xor (p and r) .
```

which would make it likely that terms containing `and` would fail to terminate. However, OBJ3's id processing will discard the problematic equations and add an id condition to the rule, so that it looks as follows:

```
eq p and (q xor r) = (p and q) xor p and r
    if not (r === false or p === true or q === false) .
```

Id conditions are normally not displayed. But when rules are displayed either in `verbose` mode, or with a `show all command`, then the id conditions are shown. For example, you can see what id completion did to `PROPC` with `id:`, including those generated by identity completion, by using the command

```
show PROPC .
```

or more specifically, the command

```
show eqs PROPC .
```

Also, when a module is processed in `verbose` mode, some details of the completion process are shown, including the new rule instances that are generated, and indications of modifications or additions to rules. This extra information can help to understand non-termination problems. The rules that are automatically added by id processing have automatically generated labels of the form "`compl`⟨*Nat*⟩".

The object `BSET` of Appendix Section C.1 illustrates a different approach, which is to explicitly make the distributive law a *conditional* equation. This approach can also be useful in many other cases. The object `IDENTICAL` in the standard prelude is used in this example; it consists of `BOOL` plus the operators `_===_` and `_=/==_`, which test for syntactic identity and non-identity, respectively.

For another example, if the following module is processed in verbose mode,

```
obj TST is
  sort A .
  ops c d e 0 1 : -> A .
  vars X Y : A .
  op _+_ : A A -> A [assoc comm id: 0] .
  eq X + Y = c .
endo
```

then OBJ3 will produce this output:

```
==========================================
obj TST
Performing id processing for rules
For rule: eq X + Y = c
  Generated valid rule instances:
  eq X + Y = c
  Generated invalid rule instances:
  eq Y = c
  eq X = c
  Modified rule: eq X + Y = c if not (Y === 0 or X === 0)
Done with id processing for rules
==========================================
```

No new rules are generated here, but an id condition is added to the given rule. A rule instance is considered invalid if its left side is a variable, or if it would "obviously" cause non-termination, e.g., if its left and right sides are the same term; such rules are discarded. The following commands

```
show rule .1 .
show all rule .1 .
```

produce the following output:

```
==========================================
show rule .1 .
rule 1 of the last module
  eq X + Y = c
==========================================
show all rule .1 .
rule 1 of the last module
  eq X + Y = c if not (Y === 0 or X === 0)
==========================================
```

The following is a somewhat more complicated example:

```
obj TST is
  protecting TRUTH-VALUE .
  sort A .
  op 0 : -> A .
  op _+_ : A A -> A [assoc id: 0] .
  op 1 : -> A .
  op _*_ : A A -> A [assoc id: 1] .
  op f : A -> A .
  ops a b c d e f : -> A .
  var X Y : A .
  eq (X * Y) + f(X * Y) = f(X) .
endo
```

Its `verbose` output is:

```
==========================================
obj TST
Performing id processing for rules
For rule: eq (X * Y) + f(X * Y) = f(X)
  Generated valid rule instances:
  eq (X * Y) + f(X * Y) = f(X)
  eq X + f(X) = f(X)
  eq Y + f(Y) = f(1)
  eq f(0) = f(1)
  Generated invalid rule instances:
```

```
  eq f(0) = f(0)
  Added rule: [compl16] eq f(0) = f(1)
  Added rule: [compl17] eq X + f(X) = f(X) if not X === 0
  Modified rule: eq (X * Y) + f(X * Y) = f(X) if not (X === 0 and Y ===
    1)
Done with id processing for rules
==========================================
```

The rule `compl16` must be added because the top operator of its left side is `f` rather than `_+_`, and OBJ3 stores rules according to the top symbol of their left side. Notice that the left side of rule `compl17` is a generalization of the original left side, and in fact, is a strict generalization because with OBJ3's built-in matching, the original rule's left side cannot match `X + f(X)`, the left side of the new rule. Therefore, it might make sense to delete the original rule; however, OBJ3 does not do this, to avoid the potential confusion of having rules given by the user disappear. The id completion process may not be correct unless the original rule set is confluent, in the sense that the set of rules available after id completion is confluent modulo OBJ3's built-in matching (which is basically associative, or associative-commutative, matching, plus some quite limited identity matching). In this example, the confluence assumption implies that it is valid not to add the rule `eq Y + f(Y) = f(1)`. Note that in this rather contrived example, the original system was not confluent. The following is a simpler example of a single rule that is not confluent:

```
eq f(X) + f(Y) = X .
```

where `+` is commutative.

Warning: Although Release 2.04 of OBJ3 handles associative rewriting in part through special internal data structures, its way of dealing with subsorts may generate new "extension" equations from given equations that involve associative operators; see [99] for further details. In many cases, these new equations are not only unnecessary, but can also greatly slow down reduction. For example, if you read in the objects `NAT` and `INT` from Appendix Section C.7 and then ask OBJ to show you the equations, among them you will find

```
 [×] eq 0 + I + ac_E = ac_E + I .
```

Although this equation can be very expensive to match, it adds nothing to the power of the equation that it extends, namely

```
eq 0 + I = I .
```

Unfortunately, the current implementation of associative rewriting does sometimes require equations like [×] to be generated, and it can be hard to tell when they are useless. The issues involved in implementing rewriting modulo associativity and identity are surprisingly subtle, and it is likely that improvements on the techniques that we have used in OBJ3 could be found; see [99, 96] for further discussion.

3.2 Opening and Closing Modules

A module can be temporarily enriched after it has been defined, by using the command

open ⟨*ModExp*⟩ .

or, for the last module, just the command

```
open .
```

This is useful for many applications, and in particular for theorem proving.

Warning: The blank and period are required here.

Exactly the same syntax can be used for adding declarations to an open module as for originally introducing them into modules; thus, operators can be added with the `op` command, sorts with the `sort` command, and so on. All other top level commands (e.g., `in`, `set`, `show`, `select`, and `do`) also work as usual.

Normally a module that has been opened should eventually be closed, using the command

```
close
```

Warning: There is no period after this command; it is considered self-delimiting.

The command `open` creates a hidden object (called "`%`") that includes the given object, and the command `close` causes the hidden object "`%`" to be deleted. All enrichments to an opened module "disappear" when it is closed. This allows an object to temporarily have more structure than when it was originally defined, which can be very useful in theorem proving examples, as illustrated many times in Section C.4.

OBJ3 separately keeps track of the "last" module and the "open" module (if any). Therefore, it is possible (for example) to show the module `INT` while the module `LIST` is open; this will make `INT` the "last" module, but all newly declared elements will still go into the open `LIST` module. The "last" module can be identified by the command "`show name .`" and the open module can be identified by "`show name open .`".

The variables declared in a module are no longer available when it is opened, but they can be made available with the command

```
vars-of .
```

The variables that are available can be seen by using the command

show vars [⟨*ModExp*⟩].

An alternate version of `open` called `openr` (to suggest "open retentive") retains additions after closure; it is closed with just `close`. This can be useful for including lemmas when OBJ3 is used for theorem proving.

Warning: If an enriched module has been incorporated into some other module, either directly (e.g., by `protecting`) or indirectly (by appearing in a module expression), then the incorporating module may no longer be valid with respect to the enriched version of the incorporated module.

The command

```
select open .
```

makes the open module the last module (i.e., the default for `show` commands, etc.). In fact, "`open`" can be used as a short name for the open module in any of the `show` commands.

Warning: If you `show` the open module, OBJ will display the name of the underlying module, but marked with "`*** open`" as a reminder.

3.3 Built-ins and the Standard Prelude

Usually, languages have some built-in data types, such as numbers and identifiers. OBJ is sufficiently powerful that it does not need built-ins, because it can define any desired data type; but building in the most frequently used data types can make a great difference in efficiency and convenience. OBJ3 has predefined objects `TRUTH-VALUE`, `TRUTH`, `BOOL`, `IDENTICAL`, `NAT`, `NZNAT`, `INT`, `RAT`, `FLOAT`, `QID`, `QIDL`, and `ID`, plus the parameterized tuple objects described in Section 4.2, the theory `TRIV` described in Section 4.1, and some other objects that serve a technical purpose.

`TRUTH-VALUE` provides just the truth values `true` and `false`, while `TRUTH` enriches `TRUTH-VALUE` with `_==_`, `_=/=_` and `if_then_else_fi`, and `BOOL` adds the expected syntax and semantics for Booleans, e.g., infix associative `and`, `or`, and `iff`, infix `implies`, prefix `not`. Indeed, the object `PROPC` of Section 2.4.6 above can be considered a *specification* for these features of the Booleans, except of course, that quoted identifiers are not provided, but `_==_`, `_=/=_` and `if_then_else_fi` are provided; also, note that these last three operators are polymorphic, in the sense that they apply to any appropriate sorts. This can help, for example, with parsing problems when a condition is of the form `E == C`, where `C` is an ambiguous constant (i.e., there is more than one constant with that name) and `E` is has a well-defined unique sort.

The object `IDENTICAL` can be used in place of `BOOL`; it provides the operators `_===_` and `_=/==_` which check for literal identity of terms without evaluating them.

`NAT`, `NZNAT`, `INT` and `RAT` provide natural numbers, non-zero naturals, integers and rationals, respectively, while `FLOAT` provides floating point numbers, each with the usual operators, having the expected attributes. You can discover exactly what any predefined object provides by using the `show` command, or else by looking at the file that defines OBJ3's predefined objects, which is `obj/lisp/prelude/obj3sp.obj` in the standard distribution of OBJ3; this file is executed as a standard prelude whenever a new instance of OBJ3 is constructed.

`QID`, `QIDL` and `ID` provide identifiers. `QID` and `QIDL` identifiers begin with the apostrophe symbol, e.g., `'a`, `'b`, `'1040`, and `'aratherlongidentifier`. `QID` has no operators, while `QIDL` and `ID` have equality, disequality, and lexicographic order, which is denoted `_<_`, and also include everything that `BOOL` does.

Warning: Data elements from `ID` lack the initial apostrophe, and therefore must be used very carefully to avoid massive parsing ambiguities.

Some of OBJ3's predefined modules were implemented by encapsulating Lisp code in objects, as can be seen by looking at the OBJ3 standard prelude. The possibility of implementing other efficient built-in data structures and algorithms remains available to sophisticated users, and has many potential applications, such as building graphics interfaces. Details of the syntax for built-in sorts and equations appear in Appendix Section D.

3.4 Files and Libraries

To reuse code, it and anything that it relies upon (its context) must be available. Files provide a convenient way to store and retrieve modules along with their contexts. The context of a given file of modules and views can be preserved by prefacing the file with a command that fetches whatever it depends upon. Thus, an OBJ file may contain modules, views, and other top level OBJ commands, including reduction commands and the `in` (or `input`) command, which reads in and executes a file. For example,

```
in mysys
```

reads the file `mysys.obj`, adding its modules and views to the current context, executing whatever commands it may contain, including nested `in` commands, and checking that a consistent context is formed as they are added to the database. This "batch mode" use of OBJ3 is in fact more convenient in practice than using it interactively.

If `eof` appears as a top level command in a file, then everything after it is ignored. This can be convenient during debugging.

Allowing files to include top level commands is very convenient and flexible. For example, after constructing a particular multi-module context, one can execute some illustrative examples. UNIX directories provide a convenient way to organize files into libraries, because a given directory can have subdirectories named by keywords, with further named subdirectories, etc. For example, the propositional calculus decision procedure may be found in file `obj/exs/propc.obj`, its test cases in `obj/exs/propc-exs.obj`, and the results of running them might be in `obj/exs/propc-exs.lg`. Note that a given file can be included in many other files, located in many different subdirectories.

The command

`cd` ⟨*Directory*⟩

can be executed at the top level of OBJ3, and will change the current directory for OBJ3 to be the given directory. The command `cd ~` can be used to change to one's home directory. A file name beginning with "`~/`" will be expanded to the user's home directory in most contexts.

The command `pwd` reports the current working directory. The command `ls` lists the files in the current working directory.

Warning: These commands do not have final periods.

3.5 Comments

OBJ has two kinds of comment, those that print when executed, and those that don't. The former are lines that begin with "`***>`", and the latter are lines that begin with "`***`". In many cases, these comment indicators can also appear part way through a line, in which case the remainder of the line is treated as a comment. "`---`" is a synonym for "`***`" for comments. The printing comments can be useful when OBJ is used in "batch mode." These comments must either be at the outer-most level or the top level of a module and cannot appear inside of other basic syntactic units; specifically, they cannot appear in terms (and hence equations) or views.

If the first non-blank character after "`***`" is a "`(`", then the comment extends from that character up to the next balancing "`)`". This makes it easy to comment out several lines at once. For example,

```
*** (
  eq X * 0 = 0 .
  eq X + X = X .
     )
```

Warning: Be careful of comments that have parentheses in them; for example, in

```
*** (This is the idempotent law:)  eq X + X = X .
```

the comment only extends to the balancing ")", and does not include the equation. This treatment is inconsistent with release 1, and may cause errors in some older specifications.

4 Parameterized Programming

Both the costs and the demands for software are enormous, and are growing rapidly. One way to diminish these effects is to maximize the **reuse** of software, through the systematic use of what we call **parameterized programming** (see [85, 47, 52, 49, 36, 37, 55, 77]). Successful software reuse depends upon the following tasks being sufficiently easy:

1. finding old parts that are close enough to what you need;
2. understanding those parts;
3. getting them to do what you need now; and
4. putting them all together correctly.

Under these conditions, the total effort, and especially debugging and maintenance effort, can be greatly reduced. Objects, theories, views and module expressions provide formal support for these tasks. The basic idea of parameterized programming is a strong form of *abstraction*: to break code into highly parameterized mind-sized pieces. Then one can construct new programs from old modules by instantiating parameters and transforming modules. Actual parameters are modules in this approach, and interface specifications include semantic as well as syntactic information.

Ada [22] generic packages provide only part of what would be most useful. In particular, Ada generic packages provide no way to document the *semantics* of interfaces, although this feature can greatly improve the reliability of software reuse and can also help to retrieve the right module from a library, as discussed in [49]. Also, Ada provides only very weak facilities for combining modules. For example, only one level of module instantiation is possible at a time; that is, one cannot build `F(G(A))`, but rather one must first define `B` to be `G(A)`, and then build `F(B)`.

Parameterized modules are the basic building blocks of parameterized programming, and its theories, views and module expressions go well beyond the capabilities of Ada generics. A **theory** defines the interface of a parameterized module, that is, the structure and properties required of an actual parameter for meaningful instantiation. A **view** expresses that a certain module satisfies a certain theory in a certain way (note that some modules can satisfy some theories in more than one way); that is, a view describes a *binding* of an actual parameter to an interface theory. **Instantiation** of a parameterized module with an actual parameter, using a particular view, yields a new module. **Module expressions** describe complex interconnections of modules, potentially involving instantiation, addition, and renaming of modules.

A useful insight (see [7]) is that programming in the large can be seen as a kind of *functional programming*, in which evaluating (what we call) a module expression is indeed a kind of expression evaluation; in particular, there are no variables, no assignments, and no effects, side or otherwise, just functions applied to arguments; this can provide a formal basis for software reuse [12]. However, there are also some significant differences between ordinary functional programming and module expression evaluation in parameterized programming, including semantic interfaces, the use of views, and evaluation *in context*, producing not just a single module, but also embedding it in its context, i.e., placing it within a graph of modules.

As an example of parameterized programming, consider a parameterized module `LEXL[X]` that provides lists of `X`s with a lexicographic ordering, where the parameter `X` can be instantiated with any partially ordered set. Thus, given `QIDL` that provides identifiers (and in particular, words) with their usual (lexicographic) ordering, then `LEXL[QIDL]` provides a lexicographic ordering on lists of words (i.e., on "phrases," such as book titles). And `LEXL[LEXL[QIDL]]` provides a lexicographic ordering on list of phrases (such as lists of book titles) by instantiating the ordering that `LEXL[X]` requires with the one that `LEXL[QIDL]` provides, namely lexicographic ordering[17]. Similarly, given a module `SORTING[Y]` for sorting lists of `Y`s (again for `Y` any partially ordered set, and assuming that `SORTING[Y]` imports `LEXL[Y]`), we can let `Y` be `LEXL[QIDL]` to get a program `SORTING[LEXL[QIDL]]` that sorts lists of book titles.

[17] In practice, the two kinds of list should use different notation, to avoid parse ambiguities, as in Section 4.5 below.

Let us examine this example a little more closely. In general, a module can define one or more data structures, with various operators among them, possibly importing some data structures and operators from other modules[18]. For example, `LEXL[X]` should define or import lists of `X`s and provide a binary relation, say `L1 << L2`, meaning that list `L1` is the same as or comes earlier in the ordering than `L2`. The interface theory for `LEXL` is `POSET`, the theory of partially ordered sets, and hereafter we will use the notation `LEXL[X :: POSET]` to indicate this. To instantiate a formal parameter with an actual parameter, it is necessary to provide a **view**, which *binds* the formal entities in the interface theory to actual ones. If there is a default view of a module `M` as a partial order, we can just write `LEXL[M]`; for example, if `M` = `QIDL`, there is an obvious view that selects the predefined lexicographic ordering relation on identifiers. `LEXL[QIDL]` in turn provides another lexicographic ordering, and a default view from `POSET` using this ordering makes it legal to write `SORTING[LEXL[QIDL]]`. (Code for this example is given in Sections 4.4 and 4.5.) The next two subsections discuss theories and views, respectively.

4.1 Theories

Theories are used to express properties of modules and module interfaces. In general, OBJ3 theories have the same structure as objects; in particular, theories have sorts, subsorts, operators, variables and equations, can import other theories and objects, and can even be parameterized. Semantically, a theory denotes a "variety" of models, containing all the (order sorted) algebras that satisfy it, whereas an object defines just one model (up to isomorphism), its initial algebra. As a result of this, theories are allowed to contain equations that are prohibited in objects, because they cannot be interpreted as rewrite rules. In particular, equations in theories may have variables in their right side and conditions that do not occur in their left side; also the left side may be a single variable. Nonetheless, Release 2 of OBJ3 allows reductions to be executed in the context of theories, and simply ignores any equations that cannot be interpreted as rewrite rules. However, the `apply` feature (described in Section 5) does fully support equational deduction with such non-rewrite-rule equations. Another difference is that built-in rules (see Appendix D.1.3) are not allowed in theories.

Now some examples, declaring some interfaces with properties that might have to be satisfied for certain modules to perform correctly. The first is the trivial theory `TRIV`, which requires nothing except a sort, designated `Elt`.

```
th TRIV is
  sort Elt .
endth
```

This theory is predefined in OBJ3, as part of the standard prelude.

The next theory is an extension of `TRIV`, requiring that models also have a given element of the given sort, here designated "`*`".

```
th TRIV* is
  using TRIV .
  op * : -> Elt .
endth
```

Of course, this enrichment is equivalent to

```
th TRIV* is
  sort Elt .
  op * : -> Elt .
endth
```

which may seem clearer. These first two theories impose only syntactic requirements.

Next, the theory of pre-ordered sets; its models have a binary infix `Bool`-valued operator `<=` that is reflexive and transitive.

```
th PREORD is
  sort Elt .
```

[18]In general, modules may have internal states; although this feature is not discussed in this paper, the reader can consult [85, 70, 55] and [74] for further information about approaches to this important topic, which is implemented in the FOOPS [74, 87], OOZE [3], and LILEANNA [85] systems.

```
  op _<=_ : Elt Elt -> Bool .
  vars E1 E2 E3 : Elt .
  eq E1 <= E1 = true .
  cq E1 <= E3 = true if E1 <= E2 and E2 <= E3 .
endth
```

Note the use of `and` in the condition of the last equation, used to express transitivity; it has been imported from `BOOL`. Recalling that previously declared variables can occur in terms submitted for reduction, the term

```
reduce E1 <= E1 .
```

reduces to `true` (after producing a warning about the presence of variables).

Similarly, the theory of partially ordered sets, with models having a binary infix `Bool`-valued operator `<` that is anti-reflexive and transitive, can be expressed as follows:

```
th POSET is
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars E1 E2 E3 : Elt .
  eq E1 < E1 = false .
  cq E1 < E3 = true if E1 < E2 and E2 < E3 .
endth
```

The theory of an equivalence relation has a binary infix `Bool`-valued operator, here denoted `_eq_`, that is reflexive, symmetric and transitive.

```
th EQV is
  sort Elt .
  op _eq_ : Elt Elt -> Bool [comm].
  vars E1 E2 E3 : Elt .
  eq (E1 eq E1) = true .
  cq (E1 eq E3) = true if (E1 eq E2) and (E2 eq E3) .
endth
```

Note the use of the `[comm]` attribute here; of course, we could instead have given the equation for symmetry,

```
eq E1 eq E2 = E2 eq E1 .
```

Finally, the theory of monoids, which in Section 4.7 will serve as an interface theory for a general iterator that in particular gives sums and products over lists.

```
th MONOID is
  sort M .
  op e : -> M .
  op _*_ : M M -> M [assoc id: e] .
endth
```

The possibility of expressing *semantic* properties, such as the associativity of an operator, as part of the interface of a module is a significant advantage for parameterized programming over traditional functional programming. For example, traditional functional programming can easily provide a (second-order) function to iterate any given binary function (such as integer addition) over lists, but it cannot express the condition that the binary function must be associative, although this property is necessary for certain optimizations to be correct (see Section 4.7 for the details of this example). There is no reason why the language used to express assertions in theories couldn't be full first-order logic (including quantifiers); in fact, this might be a very desirable extension of OBJ3. However, the expressive power of the `protecting` importation of objects into theories (which requires that the given object must be interpreted with initial algebra semantics) and in particular the use of such importations for the `BOOL` object, allows quite sophisticated theories to be defined in OBJ3 even with its current restriction to equational logic. For example, the theory `FIELD` of fields, which is well known not to be equationally definable, can easily be defined as follows[19]

[19] However, as pointed out in Section 2.3.3, general sort constraints like the one given in this theory are not yet implemented.

```
th FIELD is
  sorts Field NzField .
  subsorts NzField < Field .
  protecting BOOL .
  op 0 : -> Field .
  op 1 : -> NzField .
  op _+_ : Field Field -> Field [assoc comm id: 0].
  op _*_ : Field Field -> Field [assoc comm id: 1].
  op _*_ : NzField NzField -> NzField [assoc comm id: 1].
  op -_ : Field -> Field .
  op _^-1 : NzField -> NzField .
  op nz : Field -> Bool .
  vars X Y Z : Field .
  vars X' Y' : NzField .
  as NzField : X if nz(X) .
  eq X + (- X) = 0 .
  eq X' * (X' ^-1) = 1 .
  eq X * (Y + Z) = (X * Y) + (X + Z) .
  cq X = 0 if not nz(X) .
endth
```

4.2 Parameterized Modules

Let us now consider some parameterized modules. First, a parameterized LIST object, with TRIV as its interface theory:

```
obj LIST[X :: TRIV] is
  sorts List NeList .
  subsorts Elt < NeList < List .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil prec 9] .
  op __ : NeList List -> NeList [assoc prec 9] .
  op head_ : NeList -> Elt .
  op tail_ : NeList -> List .
  op empty?_ : List -> Bool .
  var X : Elt .
  var L : List .
  eq head(X L) = X .
  eq tail(X L) = L .
  eq empty? L = L == nil .
endo
```

Warning: The interface theories of parameterized modules must have been defined earlier in the program text. For example, TRIV must have been defined before the above LIST module; this is not a problem in this case, because TRIV is predefined.

Modules can have more than one parameter. A two parameter module has an interface with the syntax [X :: TH1, Y :: TH2], and if the two theories are the same, we can just write [X Y :: TH]. Now a parameterized theory, vector spaces over a field:

```
th VECTOR-SP[F :: FIELD] is
  sort Vector .
  op 0 : -> Vector .
  op _+_ : Vector Vector -> Vector [assoc comm id: 0] .
  op _*_ : Field Vector -> Vector  .
  vars F F1 F2 : Field .
  vars V V1 V2 : Vector .
  eq (F1 + F2) * V = (F1 * V) + (F2 * V) .
  eq (F1 * F2) * V = F1 * (F2 * V) .
  eq F * (V1 + V2) = (F * V1) + (F * V2) .
```

```
endth
```

For $2 \leq n \leq 4$, the predefined parameterized nTUPLE modules provide n-tuples, with all n interface theories being TRIV. Here is the code for $n = 2$:

```
obj 2TUPLE[C1 :: TRIV, C2 :: TRIV] is
  sort 2Tuple .
  op <<_;_>> : Elt.C1 Elt.C2 -> 2Tuple .
  op 1*_ : 2Tuple -> Elt.C1 .
  op 2*_ : 2Tuple -> Elt.C2 .
  var E1 : Elt.C1 .
  var E2 : Elt.C2 .
  eq 1* << E1 ; E2 >> = E1 .
  eq 2* << E1 ; E2 >> = E2 .
endo
```

Note the use of the qualifications in the sorts Elt.C1 and Elt.C2.

4.3 Views

A module can satisfy a theory in more than one way, and even if there is a unique way, it can be arbitrarily difficult to find. We therefore need a notation for describing the particular ways that modules satisfy theories. For example, NAT can satisfy POSET with the usual "less-than" ordering, but "divides-and-unequal" and "greater-than" are also possible; each of these corresponds to a different view. Thus, an expression like LEXL[NAT] (where LEXL has interface theory POSET, as in Section 4.1) would be ambiguous if there were not certain definite conventions for default views.

More precisely now, a **view** ϕ from a theory T to a module M, indicated $\phi\colon \mathtt{T} \Rightarrow \mathtt{M}$, consists of a mapping from the sorts of T to the sorts of M preserving the subsort relation, and a mapping from the operators of T to the operators of M preserving arity, value sort, and the meanings of the attributes assoc, comm, idem, id: and[20] idr: (to the extent that these attributes are present), such that every equation in T is true of every model of M (thus, a view from one theory to another is called a "theory interpretation" in logic [9]). The mapping of sorts is expressed with the syntax

```
sort S1 to S1' .
sort S2 to S2' .
...
```

and the mapping of operators is expressed with the syntax

```
op o1 to o1' .
op o2 to o2' .
...
```

Warning: The final blank and period are required. Comments may not appear anywhere inside of views.

Thus, the mapping of sorts and the mapping of operators can each be seen as sets of pairs. The operators o1, o1', o2, ..., may be designated by operator forms, or operator forms plus value sort and arity, possibly qualified by sort and/or module, as needed for disambiguation. As explained below, the target operators o1', o2', ..., can also be derived operators, i.e., terms with variables.

These two sets of pairs together are called a **view body**. The syntax for defining a view at the top level of OBJ3 requires giving names for the source and target modules, and (usually) a name for the view. For example,

```
view NATG from POSET to NAT is
  sort Elt to Nat .
  op _<_ to _>_ .
endv
```

defines a view called NATG from POSET to NAT, interpreting _<_ in POSET as _>_ in NAT.

Now some views that involve derived operators. First, consider

[20] For this purpose, id: and idr: are treated as equivalent.

```
view NATLEQ from PREORD to NAT is
  vars L1 L2 : Elt .
  op L1 <= L2 to L1 < L2 or L1 == L2 .
endv
```

This maps `<=` in `PREORD` to "less-than-or-equal" in `NAT`, which for illustrative purposes is defined here from `<` and `==`, even though `<=` is already defined in `NAT`. Similarly, the following view maps `<` in `POSET` to the relation "divides-and-unequal" in `NAT` (note that divides-and-unequal is a partial ordering that is not a total ordering):

```
view NATD from POSET to NAT is
  vars L1 L2 : Elt .
  op L1 < L2 to L1 divides L2 and L1 =/= L2 .
endv
```

Warning: The variables must be declared, using sorts from the source theory, while the target terms in operator pairs must be of the corresponding sorts from the target module. For parsing the target terms, the variables are re-declared with the appropriate sorts.

When the user feels that there is an obvious view to use, it can be annoying to have to write out that view in full detail. **Default views** often allow omitting views within module expressions, by capturing the intuitive notion of "the obvious view." A default view is a **null view**[21], which is the extreme case of an **abbreviated view** in which the view is abbreviated to nothing. Given a view $\phi\colon \mathtt{M} \Rightarrow \mathtt{M}'$, there are two rules for abbreviating sorts:

1. Any pair of the form `S to S` can be omitted, except in the case that the target sort `S` is a sort in a parameter theory.
2. A pair `S to S'` can be omitted if `S` and `S'` are each the principal sort of their module. The **principal sort** of a module is intuitively the first sort introduced into its body. More precisely, it is defined as follows:

   (a) the first sort explicitly declared in the module, if there is one;
   (b) or else the principal sort of the first imported module, if there is one;
   (c) or else the principal sort of the first parameter theory, if there is one;
   (d) or else `Bool`, provided that implicit importation of `Bool` has not been disenabled.

   If none of these conditional hold, then the module has no sorts!

Warning: The handling of default views described above is slightly different from that in Release 1, and some default views may be computed differently.

As a special case, every module has a default view from `TRIV` with its principal sort as the target for `Elt`. For another example, the default view from `POSET` to `NAT` is

```
view NATV from POSET to NAT is
  sort Elt to Nat .
  op _<_ to _<_ .
endv
```

In the following abbreviated view

```
view NATG from POSET to NAT is
  op _<_ to _>_ .
endv
```

the pair `Elt to Nat` has been omitted.

The sort to be used as the principal sort of a module can be explicitly set with a declaration in the module of the form

[21] Some further conventions for default views are described in [52], but these are not implemented in Release 2 of OBJ3; also, the discussion in [52] does not reflect some improvements to OBJ3 made after that paper was written.

```
principal-sort ⟨Sort⟩ .
```

where `principal sort` may be abbreviated to `psort`. Note that this does not create a sort, but just declares that a certain existing sort should be taken as the principal sort of the module. This feature is not usually needed, because the default choice of principal sort, as the first sort "mentioned", is quite good, but it is needed (for example) if the sort desired to be principal comes from an interface theory.

The commands

```
show psort [⟨ModExp⟩] .
```

and

```
show psort .
```

show the principal sort of a given module, or of the current module if none is given.

There are also conventions for omitting operator pairs from a view:

1. Any pair of the form `o to o` can be omitted.
2. If `o to o'` is a pair in a view, and if `o` and `o'` have attributes `id: e` (or `idr: e`) and `id: e'` (or `idr: e'`) respectively, then `e to e'` can be omitted.

For example, the default view from `POSET` to `NAT` has `Elt to Nat` and `_<_ to _<_`, and the default view from `MONOID` to `NAT` is

```
view NAT* from MONOID to NAT is
  sort M to Nat .
  op _*_ to _*_ .
  op e to 1 .
 endv
```

where `e` maps to `1` because the identity attribute of `*` is preserved. The following is a non-default view of `NAT` as a `MONOID`,

```
view NAT+ from MONOID to NAT is
  op _*_ to _+_ .
  op e to 0 .
endv
```

where `e to 0` could also be omitted by preservation of the identity attribute.

It can be shown [52] that if ψ is an abbreviation of each of two views ϕ and ϕ', then $\phi = \phi'$. From this, it follows that there is at most one null view from any module to any other.

Warning: Although our default view conventions work well for simple examples, it cannot be expected that they will always produce exactly the view that a user wants in more complex examples. Also, there is more going on behind the scenes than meets the eye.

Next, a view where the target module involves a parameter:

```
view LISTM from MONOID to LIST is
  op _*_ to __ .
  op e to nil .
endv
```

This view can actually be abbreviated to the following null view:

```
view LISTM from MONOID to LIST is endv
```

Warning: Although the source components of the operator mapping of a view ϕ: `M` $\to$ `M'` involve only operators from `M` itself, the target components can be terms that involve anything that `M'` imports; indeed, they do not have to involve operators from `M'` itself at all. Also notice that ϕ could be meaningless if some modules imported by `M` are not also imported by `M'`, because then the translations of some of the equations in `M` might not make sense in `M'`.

We wish to emphasize the documentation aspect of views, because it is unique to OBJ and also seems very practical. In principle, theorem proving technology could be used to verify that a given mapping really is a view, that is, that the semantic properties specified in the source theory are in fact satisfied by the target module; presumably, this could be done using OBJ itself as a theorem prover, using techniques from [51, 53], or the Kumo system [65, 64, 78, 60]. However, this often may not be worth the effort in practice, and non-verified views should be seen as documenting the programmer's intentions and beliefs about the semantic properties of modules. Going a little further, a level of assurance could be associated with a view, reflecting the degree to which it has been validated, ranging from "pious hope" to "mechanically verified theorem." An intermediate level of assurance that may often be practical is that a paper and pencil proof has been given with the usual informal rigor of mathematics. Some form of automatic testing could also be provided.

4.4 Instantiation

This subsection discusses instantiating the formal parameters of a parameterized module with actual modules. This construction requires a view from each formal interface theory to the corresponding actual module. The result of such an instantiation replaces each interface theory by its corresponding actual module, using the views to bind actual names to formal names, and avoiding multiple copies of shared submodules.

Let us consider the `make` command, which evaluates a module expression, and then adds the result to the OBJ3 database. This can be used to instantiate a parameterized module with an actual module *via* a view, and if a module name is used instead of a view, then the default view of that module from the interface theory of the parameterized module is used, if there is one. For example, given the parameterized object `LEXL[X :: POSET]`, we can write

```
make LEXL-NATG is LEXL[NATG] endm
```

using the explicit view `NATG`, while

```
make NAT-LIST is LIST[NAT] endm
```

uses the default view from `TRIV` to `NAT` to instantiate the parameterized module `LIST` with the actual parameter `NAT`. Similarly, we might have

```
make RAT-LIST is LIST[RAT] endm
```

where `RAT` is the field of rational numbers, using a default view from `TRIV` to `RAT`, or

```
make RAT-VSP is VECTOR-SP[RAT] endm
```

using the default view from `FIELD` to `RAT`. More interestingly,

```
make STACK-OF-LIST-OF-RAT is STACK[LIST[RAT]] endm
```

uses two default views. Expressions like `LEXL[NATG]`, `LIST[NAT]`, and `STACK[LIST[RAT]]` are **module expressions**, as discussed further in Section 4.5 below.

In general,

```
make M is P[A] endm
```

is equivalent to

```
obj M is protecting P[A] . endo
```

where `A` may be either a module or a view. Thus, `make` is redundant, and in fact it was not implemented in OBJ2.

If a non-null view is only used once, say to instantiate a parameterized module, it can be defined "on the fly" where it is used, with the syntax

`view to` $\langle$*ModExp*$\rangle$ `is` $\{\langle$*SortMap*$\rangle$ $|\langle$*OpMap*$\rangle\}$`...` `endv`

For example, if the view `NATG` were not already defined, then we could get the same effect from

```
make LEXL-NATG is LEXL[view to NAT is op _<_ to _>_ .  endv] endm
```

as we did from making LEXL[NATG].

A sort name (possibly preceded by the keyword sort) given as an actual parameter to a parameterized module, is treated as an abbreviation for a default view mapping the principal sort of the source theory to the given sort. For example, the object

```
obj LEXL-NAT is
  protecting NAT .
  protecting LEXL[Nat] .
endo
```

uses the sort Nat as an abbreviation for the view

```
view from POSET to NAT is
  sort Elt to Nat .
  op _<_ to _<_ .
endv
```

When the interface theory is TRIV, it is enough just to give a sort name to define a view. For example, 2TUPLE[Int,Bool] is a module expression whose principal sort consists of pairs of an integer and a truth value. A more complex example is 3TUPLE[Int,Bool,LIST[Int]].

An operator name, possibly preceded by the keyword op, given as an actual parameter to a parameterized module, is treated as an abbreviation for a default view mapping the operator of similar rank in the source theory to the given operator. For example, the module expression MAP[(sq_).FNS] is used in Section 4.7.

Sometimes it is convenient to import a module expression with its formal parameters instantiated by (some of) those of a parameterized module into which it is imported, as in

```
obj LEXL[X :: POSET] is
  protecting LIST[X] .
  op _<<_ : List List -> Bool .
  vars L L' : List .
  vars E E' : Elt .
  eq L << nil = false .
  eq nil << E L = true .
  eq E L << E L' = L << L' .
  cq E L << E' L' = E < E' if E =/= E' .
endo
```

where LIST[X] uses the default view of X as TRIV. Thus,

```
make LEXL-NAT is LEXL[NAT] endm
```

uses the default view of NAT as POSET to give a lexicographic ordering on lists of natural numbers, and also instantiates LIST with NAT. Similarly,

```
make LEXL-NATD is LEXL[NATD] endm
```

orders lists of NATs by the divisibility ordering on NATs, while

```
make PHRASE is LEXL[QIDL] endm
```

uses the lexicographic ordering on QIDL to give a lexicographic ordering on lists of identifiers, and thus (for example) on titles of books. (This example, which builds upon one from [77], is continued in the following subsection.)

Now consider the case where a view that is abbreviated to just a sort name occurs in a module, and the sort name used has been declared above it, in the same module. OBJ3 treats this harmless form of self-reference by automatically creating a submodule that contains all of the definitions in the current module, up to the module expression containing the self-reference. This newly created module is then used as the target module for the view. For example,

```
obj SELF-REF is
  sort A .
  op a : -> A .
  protecting LIST[A] .
endo
```

causes the automatic generation of a submodule containing the sort `A` and the constant `a`.

Warning: These automatically created modules are considered hidden. Although they are assigned names, these names cannot appear in user-supplied input. However, users can display them with command of the form "`show sub 2 .`" (this feature is described in Appendix Section A).

Environments for ordinary programming languages are functions from names to values (perhaps with an extra level of indirection); but OBJ environments, which we call **contexts**, must also store views, which are relationships between modules, and must record module importation relationships. Section 3 discussed the submodule inclusion relation that arises from module importation, giving rise to an acyclic graph structure. Contexts with views as edges from source to target module give a general graph structure. If submodule inclusions are also included, then the submodule hierarchy appears as a particular subgraph of this overall graph.

4.5 Module Expressions

Module expressions permit defining, constructing, and instantiating complex combinations of modules, as well as modifying modules in various ways, thus making it possible to use a given module in a wider variety of contexts. The major combination modes are instantiation (as discussed above), sum, and renaming. No other implemented language that we know has such features in the language itself.

The simplest module expressions are simple named modules, which are either the predefined data types `TRUTH-VALUE`, `TRUTH`, `BOOL`, `IDENTICAL`, `NAT`, `NZNAT`, `INT`, `ID`, `QIDL`, `ID` and `FLOAT`, or any unparameterized user-defined modules available in the current context.

Renaming uses a sort mapping and an operator mapping, to create a new module from an old one; the syntax is

`({`⟨*SortMap*⟩ `|` ⟨*OpMap*⟩`}...)`

where each map can be empty, or consist of pairs of the form `sort S to S'` or `op o to o'`, respectively, with the pairs separated by commas. A renaming is applied to a module expression postfix after `*`, and creates a new module with the syntax of the preceding module expression modified by applying the given pairs to it. For example, we can use renaming to modify the `PREORD` theory to get the theory of an equivalence relation, as follows:

```
th EQV is
  using PREORD * (op _<=_ to _eq_) .
  vars E1 E2 : Elt .
  eq (E1 eq E2) = (E2 eq E1) .
endth
```

Within a module,

`dfn` ⟨*SortId*⟩ `is` ⟨*ModExp*⟩ `.`

acts as an abbreviation for

`protecting` ⟨*ModExp*⟩ `* (sort` ⟨*PSort*⟩ `to` ⟨*SortId*⟩`).`

where ⟨*PSort*⟩ is the principal sort of ⟨*ModExp*⟩. We can use `define` as a synonym for `dfn`. This feature is put to good use in the programming language example in Appendix Section C.2.

Warning: Renaming parts of a parameterized module that is instantiated with a self-referential actual parameter (i.e., with a sort or operator that was defined earlier in an enclosing module), will also affect any other instantiations of the same parameterized module that occur earlier in the program text of that enclosing module. Therefore, one should rename parts of the parameterized module before its instantiation, rather than after. For example, one should write

```
obj NO-PROB is
  protecting LIST[INT] .
  sort A .
  op a : -> A .
  protecting (LIST * (op __ to _*_))[A] .
endo
```

instead of

```
obj TROUBLE is
  protecting LIST[INT] .
  sort A .
  op a : -> A .
  protecting LIST[A] * (op __ to _*_) .
endo
```

That is, one should use the form

$(\langle Mod\rangle$ `*` $(\langle Rename\rangle))$`[`$\langle ModExp\rangle$`]`

rather than the form

$\langle Mod\rangle$`[`$\langle ModExp\rangle$`]*(`$\langle Rename\rangle$`)`

In `NO-PROB`, the module `LIST` is renamed *before* it is instantiated with a sort declared earlier in the enclosing module; therefore the renaming does not affect the earlier `LIST[INT]`. But in `TROUBLE`, the renaming is also applied to `LIST[INT]`, because this is implicitly part of the parameter and within the scope of the renaming. (Section 4.4 discusses the automatically generated module that is involved here.)

Another important module building operator is **sum**. This has the syntax

$\langle ModExp\rangle$ `+ ... +` $\langle ModExp\rangle$

which creates a new module that combines all the information in its summands. For example, the expression `A + B` creates a module that is the same as `AB` having the following definition

```
obj AB is
   pr A .
   pr B .
endo
```

The module that results from a sum of other modules is considered an extension of its summands. An important issue here is the treatment of submodules that are imported by more than one summand; for example, in `A + B`, both `A` and `B` may protect or extend `BOOL`, `NAT`, `INT` or other modules; such multiply imported modules should be *shared*, rather than repeated.

Earlier versions of OBJ had an **image** transformation with capabilities of renaming and instantiation; but because it did not use theories to describe interfaces, it now seems somewhat undisciplined, and has been abandoned, even though it can be given a respectable semantics using colimits [77]. Clear [8, 10] had a construction to *enrich* a given module, but this would be redundant in OBJ, because we need only import the given module into a new module, and then add the desired sorts, operators and equations.

Let us now continue the lexicographic ordering example from the previous section. To be able to sort lists of book titles, we might want to form something like

```
make PHRASE-LIST is LEXL[LEXL[QIDL]] endm
```

which extends the lexicographic ordering on phrases, from `LEXL[QIDL]`, to a lexicographic ordering on lists of phrases. However, this may not work quite as expected, because the two instances of list have exactly the same syntax, and thus, for example, we could not tell whether `'a 'b` was a single phrase, or a list of two phrases. We can overcome this difficulty by renaming the append constructor of one of the lists, for example, as follows:

```
make PHRASE-LIST is LEXL[LEXL[QIDL]*(op __ to _._)] endm
```

Then the two cases `('a)('b)` and `('a . 'b)` are clearly different; for a more complex phrase list, consider `('a . 'b)('c . 'd . 'e)`.

We can carry this example a bit further by giving a naive sorting algorithm for lists over a partial order `<` that relies on the power of associative rewriting. Correct behavior for such a sorting algorithm requires that, given a list `L` such that `a` and `b` occur in `L` with `a < b`, then `a` occurs earlier than `b` in the list `sorting(L)`, i.e., the list `a b` should be a sublist of `sorting(L)`; but if neither `a < b` nor `b < a`, then `a` may occur either before or after b. A tricky point about the code below is its use of a conditional equation to define the predicate `unsorted` by searching for a counter-example; note that `unsorted` does not reduce to `false` when it is false, but merely fails to reduce to `true`; this works because it is used in the condition of the equation. (Strictly speaking, this example extends `BOOL`, but we do not bother to say so explicitly, because we do not need to generate any new E-strategies or rules; instead, we just rely on the default `extending` importation of `BOOL`.)

```
obj SORTING[X :: POSET] is
  protecting LIST[X] .
  op sorting_ : List -> List .
  op unsorted_ : List -> Bool .
  vars L L' L'' : List .   vars E E' : Elt .
  cq sorting L = L if unsorted L =/= true .
  cq sorting L E L' E' L'' = sorting L E' L' E L'' if E' < E .
  cq unsorted L E L' E' L'' = true if E' < E .
endo
```

Now some test cases, and some more complex module expressions:

```
reduce in SORTING[INT] : unsorted 1 2 3 4 .  ***> should not be: true
reduce unsorted 4 1 2 3 .                    ***> should be: true
reduce sorting 1 4 2 3 .                     ***> should be: 1 2 3 4

make SORTING-PH-LIST is SORTING[LEXL[QIDL]*(op __ to _._)] endm
reduce sorting (('b . 'a)('a . 'a)('a . 'b)) .
***> should be: ('a . 'a)('a . 'b)('b . 'a)

reduce in SORTING[NATD] : sorting(18 6 5 3 1) . ***> should contain: 1 3 6 18
```

The last comment means that the list `1 3 6 18`, which is sorted by divisibility should appear as a sublist of the result of reducing `sorting(18 6 5 3 1)`; the location of `5` is not determined. It is perhaps also worth noting that the conditional equation

```
cq sorting L E L' E' L'' = sorting L E' L' E L'' if E' < E .
```

can often be matched against a given list in many different ways; some of these may succeed and others may fail. Also, note that the above code makes use of the fact that the command

```
reduce in SORTING[INT] : unsorted 1 2 3 4 .
```

changes the module in focus from the parameterized `SORTING` module to its instance `SORTING[INT]`.

A bubble sorting algorithm can be obtained from the above code by replacing the key equation

```
cq sorting L E L' E' L'' = sorting L E' L' E L'' if E' < E .
```

by its special case

```
cq sorting L E E' L'' = sorting L E' E L'' if E' < E .
```

which is obtained by letting `L'` = `nil`. However, this new program only works correctly when the actual parameter satisfies the stronger condition of being *totally ordered*; a theory for totally ordered sets may be obtained from the theory `POSET` of partially ordered sets by adding the equation

```
cq E1 < E2 or E2 < E1 = true if E1 =/= E2 .
```

In this setting, we can directly define a predicate `sorted` that evaluates to `true` when its argument is sorted.

This example illustrates the importance of semantic conditions for module interfaces: the bubble sorting algorithm is *only valid for total orders*, and if a user insists on instantiating it with an ordering that is only partial, then it may give incorrect results; this is illustrated below. (Recall that OBJ does not enforce the correctness of such user beliefs, but only allows them to be recorded in a systematic manner.) Now the code:

```
th TOSET is
  using POSET .
  vars E1 E2 E3 : Elt .
  cq E1 < E2 or E2 < E1 = true if E1 =/= E2 .
endth

obj BUBBLES[X :: TOSET] is
  protecting LIST[X] .
  op sorting_ : List -> List .
  op sorted_ : List -> Bool .
  vars L L' L'' : List .
  vars E E' : Elt .
  cq sorting L = L if sorted L .
  cq sorting L E E' L'' = sorting L E' E L'' if E' < E .
  eq sorted nil = true .
  eq sorted E = true .
  cq sorted E E' L = sorted E' L if E < E' or E == E' .
endo
```

The following illustrates correct and incorrect behavior for `BUBBLES`, and also introduces a new module expression that allows naming module expressions, with the following syntax,

⟨*Id*⟩ `is` ⟨*ModExp*⟩

which can be used inside of modules, as well as inside a `reduce` command, as below. This expression creates an "alias" for the module corresponding to the module expression.

```
reduce in A is BUBBLES[NAT]  : sorting(18 5 6 3) . ***> should be: 3 5 6 18
reduce sorting(8 5 4 2) .                          ***> should be: 2 4 5 8
reduce in B is BUBBLES[NATD] : sorting(18 5 6 3) . ***> mightnt contain: 3 6 18
reduce sorting(8 5 4 2) .                          ***> mightnt contain: 2 4 8
reduce in C is SORTING[NATD] : sorting(18 5 6 3) . ***> should contain: 3 6 18
reduce sorting(8 5 4 2) .                          ***> should contain: 2 4 8
reduce in A : sorting(9 6 3 1) .                   ***> should be: 1 3 6 9
reduce in B : sorting(9 6 3 1) .                   ***> mightnt be: 1 3 6 9
reduce in C : sorting(9 6 3 1) .                   ***> should be: 1 3 6 9
```

Here the first, second and seventh reductions are done in the context of `BUBBLES[NAT]`, while the third, fourth and eighth are done in the context of `BUBBLES[NATD]`. In fact, executing the reductions in `B` shows that violating the interface theory really can lead to incorrect results.

4.6 Top-Down Development

It might seem at first that parameterized programming is limited to a bottom-up development style, but in fact, there are many ways to realize a top-down style using OBJ:

1. Write a theory T that describes some desired behavior, and then write a module M with a view $T \Rightarrow M$. Here M may be either an object or another theory.

2. Write a module that realizes the desired behavior if the right modules are imported; write "stubs" (i.e., skeletal code) for the modules to be imported, and then elaborate them later (see [36, 37, 34]). One may be able to use the interface theories themselves as "stubs," because reductions can be executed over OBJ3 theories.

3. Write a parameterized module that realizes the desired behavior if its parameters are instantiated according to their interface theories. Then later, write modules that satisfy these interface theories, and do the instantiation.

Of course, a given step of top-down development could involve any two, or even all three of these strategies, and any number of steps can be taken. In addition, it could be useful to combine views, using the same operators that we have discussed for combining modules; however, this is not supported in Release 2 of OBJ3.

4.7 Higher-Order Programming without Higher-Order Functions

Higher-order logic seems useful in many areas, including the foundations of mathematics (e.g., type theory [111]), extracting programs from correctness proofs of algorithms, describing proof strategies (as in LCF tactics [89]), modeling traditional programming languages (as in Scott-Strachey semantics [139]), and studying the foundations of the programming process. One important advantage of higher-order programming over traditional imperative programming is its capability for structuring programs (see [94] for some cogent arguments and examples).

However, a language with sufficiently powerful parameterized modules *does not need* higher-order functions. We do not *oppose* higher-order functions as such; however, we do claim that higher-order functions can lead to code that is very difficult to understand, and that higher-order functions should be avoided where they are not necessary. We further claim that parameterized programming provides an alternative basis for higher-order programming that has certain advantages. In particular, the following shows that typical higher-order functional programming examples are easily coded as OBJ3 programs that are quite structured and flexible, and are rigorously based upon a logic that is *only first-order* and does not require reasoning about functions. The use of first-order logic makes programs easier to understand and to verify. Moreover, OBJ can use theories to document any semantic properties that may be required of functions.

One classic functional programming example is motivated by the following two instances: `sigma` adds a list of numbers; and `pi` multiplies such a list. To encompass these and similar examples, we want a function that applies a binary function recursively over suitable lists. Let's see how this looks in vanilla higher-order functional programming notation. First, a polymorphic list type is defined by something like

```
type list(T) = nil + cons(T,list(T))
```

and then the function that we want is defined by

```
function iter : (T -> (T -> T)) -> (T -> (list(T) -> T))
axiom iter(f)(a)(nil) => a
axiom iter(f)(a)(cons(c,lst)) => f(c)(iter(f)(a)(lst))
```

so[22] that we can define our functions by

```
sigma(lst) => iter(plus)(0)(lst)
pi(lst) => iter(times)(1)(lst)
```

For some applications of `iter` to work correctly, `f` must have certain *semantic* properties. For example, if we want to evaluate `pi(lst)` using as few multiplications as possible and/or as much parallelism as possible (by first converting a list into a binary tree, and then evaluating all the multiplications at each tree level in parallel), then `f` must be *associative*. Associativity of `f` implies the following "homomorphic" property, which is needed for the correctness proof of the more efficient evaluation algorithm,

```
(H) iter(f)(a)(append(lst)(lst')) = f(iter(f)(a)(lst))(iter(f)(a)(lst')
```

where `lst` and `lst'` have the same type, `list(T)`. Furthermore, if we want the empty list `nil` to behave correctly in property (H), then `a` must be an identity for `f`.

Now let's do this example in OBJ3. First, using mixfix syntax `_*_` for `f` improves readability somewhat; but much more significantly, we can use the interface theory `MONOID` to assert associativity and identity axioms for actual arguments of a generic iteration module,

```
obj ITER[M :: MONOID] is
  protecting LIST[M] .
  op iter : List -> M .
  var X : M .   var L : List .
  eq iter(nil) = e .
  eq iter(X L) = X * iter(L) .
endo
```

[22] Most people find the rank of `iter` rather difficult to understand. It can be simplified by uncurrying with products, and convention also permits omitting some parentheses; but these devices do not help much. Actually, we feel that products are more fundamental than higher-order functions, and that eliminating products by currying can be misleading and confusing.

where e is the monoid identity. Note that LIST[M] uses the default view from TRIV to MONOID. (This code uses an associative List concatenation, but it is also easy to write code using a cons constructor if desired.) Notice that all of the equations involved here are first-order.

We can now instantiate ITER to get our two example functions[23]. First,

```
make SIGMA is ITER[NAT+] endm
```

sums lists of numbers; for example,

```
reduce iter(1 2 3 4) .
```

yields 10. Similarly,

```
make PI is ITER[NAT*] endm
```

multiplies lists of numbers, and so

```
reduce iter(1 2 3 4) .
```

now yields 24. Of course, we could use renaming to get functions that are literally named sigma and pi. For example, the following module provides *both* sigma and pi.

```
make SIGMA+PI is ITER[NAT+]*(op iter to sigma) + ITER[NAT*]*(op iter to pi) endm
```

(Incidentally, this is a nice example of a complex module expression.)

Any valid instance of ITER has the property (H), which in the present notation is written simply

```
iter(L L') = iter(L) * iter(L')
```

and it is natural to state this fact with a theory and a view, as follows:

```
th HOM[M :: MONOID] is
  protecting LIST[M] .
  op h : List -> M .
  var L L' : List .
  eq h(L L') = h(L) * h(L') .
endth

view ITER-IS-HOM[M :: MONOID] from HOM[M] to ITER[M] is endv
```

This view is parameterized, because property (H) holds for all instances; to obtain the appropriate assertion for a given instance ITER[A], just instantiate the view with the same actual parameter module A.

Warning: Release 2 of OBJ3 does not implement parameterized views.

Because semantic conditions on argument functions cannot be stated in a conventional functional programming language, all of this would have to be done *outside* of such a language. But OBJ3 can not only assert the monoid property, it can even be used to *prove* that this property implies property (H), using methods first described in [51, 53] and further developed in [66, 59].

Many researchers have argued that it is much easier to use type inference for higher-order functions to get such declarations and instantiations automatically. However, the notational overhead of encapsulating a function in a module is really only a few keywords, and the appropriate definitions could even be generated automatically by a structural editor from a single keystroke; moreover, this overhead is often shared among several function declarations. Also, the overhead due to variable declarations could be reduced to almost nothing by two techniques: (1) let sort inference give a variable the highest possible sort; and (2) declare sorts "on the fly" with a qualification notation. Our position has been that the crucial issue is to make the *structure* of *large programs* as clear as possible; thus, tricks that slightly simplify notation for small examples are of little importance, and are of negative value if they make it harder to read large programs. Consequently, we have not implemented such tricks in OBJ3, because explicit declarations can save *human* program readers much effort in doing type inference.

On the other hand, our notation for instantiation can often be significantly simplified, particularly in cases where non-default views are needed, or where renaming is needed to avoid ambiguity because there is more than one instance of some module in a given context. For example,

[23] The views used below were defined in Section 4.3 above; however, a default view could be used for making PI.

```
make ITER-NAT is ITER[view to NAT is op _*_ to _+_ . endv] endm
```

is certainly more complex than `iter(plus)(0)`. However, in OBJ3 just `ITER[(_+_).NAT]` denotes the same module[24] because default view conventions map `Elt` to `Nat` in `NAT`, and `e` to `0`.

For a second example, let us define the traditional function `map`, which applies a unary function to a list of arguments. Its interface theory requires a sort and a unary function on it (more generally, we could have distinct source and target sorts, if desired).

```
th FN is
  sort S .
  op f : S -> S .
endth

obj MAP[F :: FN] is protecting LIST[F] .
  op map : List -> List .
  var X : S .   var L : List .
  eq map(nil) = nil .
  eq map(X L) = f(X) map(L) .
endo
```

Now we can instantiate `MAP` in various ways. The following object defines some functions to be used in examples below:

```
obj FNS is
  protecting INT .
  ops (sq_)(dbl_)(_*3) : Int -> Int .
  var N : Int .
  eq sq N = N * N .
  eq dbl N = N + N .
  eq N *3 = N * 3 .
endo
```

Now some reductions in objects using some non-default in-line views:

```
reduce in MAP[(sq_).FNS] : map(0 nil 1 -2 3) .  ***> should be: 0 1 4 9
reduce in MAP[(dbl_).FNS] : map(0 1 -2 3) .     ***> should be: 0 2 -4 6
reduce in MAP[(_*3).FNS] : map(0 1 -2 nil 3) .  ***> should be: 0 3 -6 9
```

In [51] there is a complete proof that the n^{th} element of `map(list)` is `f(e)`, where `e` is the n^{th} element of `list`, using OBJ3 itself as a theorem prover.

The following module does another classical functional programming example, applying a given function twice; some instantiations are also given.

```
obj 2[F :: FN] is
  op xx : S -> S .
  var X : S .
  eq xx(X) = f(f(X)) .
endo

reduce in 2[(sq_).FNS] : xx(3) .                 ***> should be: 81
reduce xx(4) .                                   ***> should be: 256
reduce in 2[(dbl_).FNS] : xx(3) .                ***> should be: 12
reduce in 2[2[(sq_).FNS]*(op xx to f)] : xx(2) .  ***> should be: 65536
reduce xx(3) .                                   ***> should be: 43046721
```

[24] We could go a little further and let `iter[(_+_).NAT]` actually denote the `iter` function on naturals, with the effect of creating the module instantiation that defines it, unless it is already present. Indeed, this is essentially the same notation used in functional programming, and it avoids the need to give distinct names for distinct instances of `iter`. This "abbreviated operator notation" could also be used when there is more than one argument, as well as for sorts. In Release 2 of OBJ3 one can get much the same effect by using qualified references to operators, as illustrated in this section.

Let us consider these examples more carefully. Because `xx` applies `f` twice, the result function `xx` of the first instantiation applies `sq_` twice, i.e., it raises to the 4th power; then the next to last instantiation applies that twice, i.e., it raises to the 16th power.

To summarize, the difference between parameterized programming and higher-order functional programming is essentially the difference between programming in the large and programming in the small. Parameterized programming does not just combine functions, it combines modules. This parallels one of the great insights of modern abstract algebra, that in many important examples, functions should not be considered in isolation, but rather in association with other functions and constants, with their explicit sources and targets, plus the equations that they satisfy. Thus, the invention of abstract algebras (for vector spaces, groups, etc.) parallels the invention of program modules (for numbers, vectors, windows, etc.); parameterized programming makes this parallel more explicit, and also carries it further, by introducing theories and views to document semantic conditions on function arguments and on module interfaces, as well as to assert provable properties of modules (such as the property (H) above). As we have already noted, it can be more convenient to combine modules than to compose functions, because a single module instantiation can compose many conceptually related functions at once, as in the complex arithmetic (`CPXA`) example mentioned near the beginning of Section 1. On the other hand, the notational overhead of theories and views is excessive for applying just one function. However, this is exactly the case where our abbreviated operator notation can be used to advantage.

We should also note that it can be much more difficult to reason with higher-order functions than with first-order functions; in fact, the undecidability of higher-order unification means that it will be very difficult to mechanize certain aspects of such reasoning[25]. Reasoning about first-order or higher-order functions can each take place in either first-order or higher-order logic. The simplest case is first-order functions with first-order logic, and this is the case that parameterized programming focuses upon. Also, it is much easier to compile and interpret first-order programs. It is worth noting that Poigné [132] has found some significant difficulties in combining subsorts and higher-order functions, and we hope to have been convincing that subsorts are very useful; however, see also [110] where significant progress has been made. Finally, note the experience of many programmers, and not just naive ones, that higher-order notation can be difficult to understand and to use.

What can we conclude from all this? We might conclude that it is better to "factorize" code with parameterized modules than with higher-order functions, and in fact, that it is better to avoid higher-order functions whenever possible. From this, one could conclude that the essence of functional programming cannot be the use of higher-order functions, and therefore it must be the lack of side effects. However, the true essence of functional programming may well reside in its having a solid basis in equational logic, because this not only avoids side effects, but also allows the kind of equational reasoning about programs and transformations that is needed to support powerful programming environments.

4.8 Hardware Specification, Simulation and Verification

This subsection develops a computer hardware verification example. The crucial advantage of using a logical programming language is that the reductions really are *proofs*, because the programs really are logical theories and computation really is deduction. This code uses the propositional calculus decision procedure object (from Section 2.4.6), thus providing an excellent example of software reuse, because `PROPC` was written in OBJ1 by David Plaisted in 1982, years before we thought of using it for hardware verification [77]. Next, `Time` is defined for use in input and output streams, which are functions from `Time` to `Prop`. An interface theory `WIRE` is defined, and then a `NOT` gate using it. The object `F` introduces two "symbolic variables," `t` and `f0`, which are a "generic" time and input stream, respectively. Finally, two `NOT` gates are composed and instantiated with `F`. Evaluating expressions in this context corresponds to proving certain theorems, in this case that the double `NOT` gate acts on input streams as a two unit delayor.

```
obj TIME is
  sort Time .
  op 0 : -> Time .
  op s_ : Time -> Time .
endo
```

[25]Similar difficulties arise for first-order unification modulo equations, such as those for the combinators, so this difficulty is not particular to higher-order logic. It is worth noting that higher-order specifications can be implemented by higher-order rewriting [97].

```
th WIRE is
  protecting TIME + PROPC .
  op f1 : Time -> Prop .
endth

obj NOT[W :: WIRE] is
  op f2 : Time -> Prop .
  var T : Time .
  eq f2(0) = false .
  eq f2(s T) = not f1(T) .
endo

obj F is
  extending TIME + PROPC .
  op t : -> Time .
  op f0 : Time -> Prop .
endo

make 2NOT is NOT[NOT[F]*(op f2 to f1)] endm
reduce f2(s s t) iff f0(t) .  ***> should be: true
```

See [51] for a precise statement of the theorem proved here, as well as a detailed justification that the given reduction really proves that theorem. Parameterized modules make this code much more readable than it would be otherwise. The same techniques seem effective for much more complex examples of hardware specification, simulation and verification, and many more examples are given in [51] and [53], along with supporting theory. The application of 2OBJ [81] to hardware verification is described in [145] and [146].

5 Applying Rules

Release 2 of OBJ3 allows users to apply rules one at a time to a given term, either "forwards" or "backwards" (i.e., either replacing an instance of the left side by the corresponding instance of the right side, or else *vice versa*). This capability is needed for many common examples of equational reasoning; for example, Appendix Section C.4.5 gives a proof from group theory that requires applying equations backwards.

The syntax is necessarily somewhat complex, as an *action* (which may be a rule), an optional *substitution*, a *range*, and a *selected subterm* may be involved. Each of these elements is discussed below. The following is an overview of the syntax:

`apply` ⟨*Action*⟩ [⟨*Substitution*⟩] ⟨*Range*⟩ ⟨*Selection*⟩ `.`

5.1 `start` and `term`

The `start` command introduces a term to which rules can then be applied; its syntax is

`start` ⟨*Term*⟩ `.`

The system keeps track of the current term, called `term`, which is either the result of the last term reduction, the term given in the last `start` command, or else is undefined. It is used as the focus for the controlled application of rules. The value of the current term can be seen by using the command

```
show term .
```

This command can also be used to see the structure of `term` in greater detail if the `print with parens` mode is `on`.

A variant of the `let` feature allows the user to give a more permanent name to `term`, using the syntax

`let` ⟨*Sym*⟩ [`:` ⟨*Sort*⟩] `= .`

The current module must be open for this to be effective.

5.2 Actions

An action is a request to print a selected subterm, to reduce it, or to apply a selected rule (possibly backwards) to it. The following shows some of the syntax:

reduction | red | print | [-] [⟨*ModId*⟩].[⟨*Nat*⟩ | ⟨*Id*⟩]

The action `reduction` (or equivalently, `red`) calls for full reduction of the selected term. The action `print` displays the selected subterm. The last action requests applying (possibly backwards) the ⟨*Nat*⟩th rule, or else the rule having ⟨*Id*⟩ as one of its labels, (optionally) from the module ⟨*ModId*⟩. Because ⟨*ModId*⟩ must be a simple module name, abbreviated module names can be very useful here. If no ⟨*ModId*⟩ is given, then the currently selected module is used.

Warning: If more than one rule has the same label, and you try to apply that label, then a warning is issued, no rule is applied, and control is returned to the top level.

There are two special actions for dealing with the built-in retract rules,

ret | -retr with sort ⟨*Sort*⟩

The first action tries to apply a built-in retract rule of the form

```
r:A>B(X) = X
```

where `X` has sort `B`, and the second action allows introducing a retract by applying this rule backwards.

5.3 Substitutions

The following rule has a variable in its left side that does not appear in its right side,

```
eq X * 0 = 0 .
```

so that its backwards version is

```
eq 0 = X * 0 .
```

which has a variable in its right side that does not appear in its left side. In such cases, it is necessary to specify a binding for the variables not in the left side (`X` in this case) in order to be able to apply the rule backwards. If these variables are not instantiated, then they are just copied into the current term. It is also natural to allow substituting variables in all cases, including forward rule applications; this is especially convenient when proving equations in the context of a theory.

A substitution gives terms as bindings for some variables in the form of a list of equations separated by commas (after the rule specification and delimited by `with`). More precisely, the syntax is

with ⟨*Var*⟩ = ⟨*Term*⟩ {, ⟨*Var*⟩ = ⟨*Term*⟩}...

where each variable mentioned must appear somewhere in the rule.

Warning: No warning is given if no binding is given for some variable that appears in the right side but not the left side of the rule. However, this condition can be detected, because a variable will have been introduced into the resulting term.

A substitution given to `print` or `reduce` is ignored.

5.4 Selecting a Subterm

There are three basic kinds of selection: for an *occurrence*, a *subsequence* (used for associative operators), and a *subset* (used for associative commutative operators).

The syntax for occurrence selection is

(⟨*Nat*⟩...)

Starting from a given term, occurrence selection progressively selects the argument positions specified by the given numbers, where both subterms and argument positions are numbered from 1. For example, if the term is `(a + (c * 2))`, then the occurrence `(2 1)` selects the subterm `c`. The empty list of numbers `()` is a selector, and it selects the whole term.

Subsequence selection has the two forms

[⟨*Nat*⟩ `..` ⟨*Nat*⟩] | [⟨*Nat*⟩]

where blanks are required around the "`..`". This kind of selection is only appropriate for terms whose top operator is associative (or associative and commutative). For such operators, a tree of terms formed with that operator is naturally viewed as the sequence of the terms at the leaves of this tree. Selecting `[k]` is the same as selecting `[k .. k]`, and it selects the `k`th subterm from the sequence (it does not form a sequence of length one). The form [⟨*Nat*⟩ `..` ⟨*Nat*⟩] forces restructuring the term so that the specified range of terms form a proper subterm of the whole term, and then it selects that term as the next current subterm. This implies that a selection may change the structure of the term, and hence that a `print` could affect the structure of the term. For example, if the current module is `INT`, and the current term (when fully parenthesized) is "`(1 * (2 * (3 * (4 * 5))))`", then the command

```
apply print at [2 .. 4] .
```

will cause term "`(1 * ((2 * (3 * 4)) * 5))`" to be printed.

Subset selection has the syntax

{⟨*Nat*⟩ [`,` ⟨*Nat*⟩]...}

where "{" and "}" are not syntactic meta-notation, but rather stand for the corresponding characters. No blanks are required in this notation. This kind of selection is only appropriate for terms with top operators that are associative and commutative. Repetitions of numbers in the list are ignored. This selector forces the given subset of the list (or more properly "bag") of terms under the top operator to be a proper subterm, and then selects that term as the next current subterm. The order of the subterms within "{}"s affects the order of appearance of these terms in the selected subterm. For example, if the current module is `INT`, and the current term (when fully parenthesized) is "`(1 * (2 * (3 * (4 * 5))))`", then the command

```
apply print at 1,3,5 .
```

causes term structure "`((1 * (3 * 5)) * (2 * 4))`" to be printed.

You can specify the top or whole of the current term by using either of the selectors `term` or `top`. It only makes sense to use these once, and often they can be omitted. One of these is required if there is no other selector, but the selector `()` could be used instead.

Selectors can be composed by separating them by `of`, as in

```
{3,1,2} of [4] of (2 3 1) of [2 .. 5] of (1 1) of term
```

Such a composition is interpreted like functional composition: the selection on the right is done first, then the second one on the result of that selection, and so on, until finally the selector on the left is done. Note that this order is the opposite of that used for the elements of an occurrence, such as `(2 1)`.

5.5 The Apply Command

The form of an apply command is `apply` followed (in order) by the action, possibly a substitution, `within` or `at`, and a composition of selectors:

`apply` { `reduction` | `red` | `print` | `retr` | `-retr with sort` ⟨*Sort*⟩ |
    ⟨*RuleSpec*⟩ [`with` ⟨*VarId*⟩ `=` ⟨*Term*⟩{`,` ⟨*VarId*⟩ `=` ⟨*Term*⟩}...] }
  { `within` | `at` }
  ⟨*Selector*⟩ {`of` ⟨*Selector*⟩}...

(Here, "{" and "}" are used for syntactic grouping.)

A user can either request a rule to be applied exactly "`at`" a selected subterm (as described the next subsection), or else to be applied "`within`" a selected subterm. In the latter case, the indicated rule is at most once. Reduction and printing always act on a whole subterm.

Sometimes, giving a substitution may make it possible to apply a rule without specifying any specific subterm, by using `within` as the range.

The resulting value of the current term is always printed after an apply command has been performed. Here are some examples of rule applications.

```
apply G.1 at term .
apply -G.1 at term .
apply -G.2 with X = a at term .
apply print at term .
apply reduction at (2 1) .
apply G.1 at () .
apply X.3 at {2} .
apply X.3 at {3,1,2} .
apply G.2 at [2 .. 4] .
apply G.1 at [2] .
apply X.1 at {2,4} of [4] of (2 2) .
apply X.1 at {2,4} of [4 .. 4] of (2 2) of top .
```

The command

```
apply ? .
```

shows a summary of the apply command syntax.

5.6 Conditional Rules

Applying conditional rules in general requires shifting the focus of reduction to the (instantiated) condition of a rule, so that rules can be applied to it. This is done by using a stack of pending actions, pushing a rule application onto the stack if its left side matches, but it has a condition the evaluation of which is still pending. When a condition has been reduced to "`true`," then the pending rule application is executed, and focus shifts to its result term. If the condition reduces to "`false`," then the rule is not applied, and focus returns to the previous term.

It is possible to request that conditions of conditional equations be directly reduced, using the command

```
set reduce conditions on .
```

The default behavior can then be restored by the command

```
set reduce conditions off .
```

(Either all nontrivial conditions must be evaluated by hand, or else none.) One reason to prefer evaluating conditions directly is that, if the top operator of the left side is associative and/or commutative, then when the rule is applied, all possible matches are tested against the condition until a successful case is found; but with controlled application, only one match is attempted. However, all potential matches can be specified by using the selection notation.

Here is a small example:

```
obj X is sort A .
  pr QID .
  subsort Id < A .
  op f : A -> A .
  var X : A .
  cq f(X) = f(f(X)) if f(X) == 'a .
  eq f('b) = 'a .
endo
```

for which the following is an output trace, illustrating how conditional rule application works:

```
==========================================
start f('b) .
==========================================
apply X.1 at term .
shifting focus to condition
condition(1) Bool: f('b) == 'a
==========================================
apply X.2 within term .
```

```
condition(1) Bool: 'a == 'a
==========================================
apply red at term .
condition(1) Bool: true
condition is satisfied, applying rule
shifting focus back to previous context
result A: f(f('b))
==========================================
```

Note that when actions are pending, "`condition`" is printed instead of "`result`," and the number of conditions being reduced (i.e., the number of pending actions) is printed in parentheses.

If you are evaluating a condition and want to force either success or failure, then you can use the following commands:

```
start true .
start false .
```

For example, the above example could have continued from "`apply X.1 at term`" with

```
==========================================
start false .
condition(1) Bool: false
condition is not satisfied, rule not applied
shifting focus back to previous context
result A: f('b)
==========================================
```

This can be used to abandon reductions that you no longer wish to perform. Note that using "`start true .`" can easily produce incorrect results, i.e., results that are not sound under order sorted equational deduction. Also, you cannot perform a controlled reduction in the middle of doing another one, and then continue the first reduction, because a new `start` causes the current state of the previous term to be lost.

The command

```
show pending .
```

shows details about the terms, rules, conditions, and replacements that are currently pending. The following is sample output from this command:

```
pending actions
  1| in f('b)  at top
   | rule cq f(X) = f(f(X)) if f(X) == 'a
   | condition f('b) == 'a  replacement f(f('b))
     2| in f('b) == 'a  at f('b)
      | rule cq f(X) = f(f(X)) if f(X) == 'a
      | condition f('b) == 'a  replacement f(f('b))
        3| in f('b) == 'a  at f('b)
         | rule cq f(X) = f(f(X)) if f(X) == 'a
         | condition f('b) == 'a  replacement f(f('b))
```

If you use the range specification `within` and the rule is conditional, then the rule will be applied at most once, and a warning will be issued like the following:

```
applying rule only at first position found: f('b)
```

6 Discussion and Conclusions

Although we are very fond of OBJ, and believe that it introduced some valuable new ideas, we certainly do not wish to claim that it is the perfect language for all applications. In particular, Release 2 of OBJ3 has a number of limitations of which we are aware:

1. OBJ3 is not a compiler, but is rather closer to an interpreter.

2. Associative/commutative rewriting can be very inefficient for large terms; but this is an inevitable result of AC matching being an NP-complete problem.

3. The parser applies precedence and gathering information *a priori*, and thus may fail on some terms that in fact are well formed.

4. Release 2 of OBJ3 gives warnings about meaningless or erroneous input, but these warning are less comprehensive than some might desire, and are not always easy to interpret.

5. OBJ3 does not have so-called "logical variables," the values for which are supplied by the system through "solving" systems of constraints, although its extension to Eqlog does [71, 72, 25].

6. OBJ3 does not keep track of whether a given equation has the status of an assumption, a conjecture, or a proven conclusion, as a good theorem proving environment should do. Similar comments apply to the ability to undo changes to modules and many other features; on the other hand, OBJ3 was not initially conceived as a theorem prover. The Kumo system [65, 64, 78, 60] was developed for this purpose, and does provide such capabilities; in addition, it generates web-based documentation for its proofs.

On the positive side, Release 2 of OBJ3 lets you experiment with a combination of parameterized programming, subsorts, rewriting modulo attributes, E-strategies, and memoization, which together allow styles that are quite different from conventional languages. We hope that you will enjoy this!

OBJ3 also provides a very workable platform for implementing other system; in particular, it has been used to implement the combined functional and object oriented system FOOPS [74], the object oriented specification language OOZE [3], a generic (i.e., metalogical) theorem prover called 2OBJ [81], and the combined logic and functional language Eqlog [71]. FOOPS and OOZE support objects with states (where we mean "objects" in the sense of object oriented programming, rather than in the sense of OBJ), and Eqlog has logical variables. OBJ3 terms can have state in some sense, but they can only be used through built-in equations. An alternative approach to object-oriented programming, supporting concurrent objects and general concurrent systems programming, is being developed at SRI in the language Maude [112, 16], which uses *rewriting logic* [113, 114] as a basis for very direct systems modeling. Someday, we may implement FOOPlog [74] or MaudeLog, which combine the functional, logic, and object oriented paradigms. We find that it is relatively easy to build such systems on top of OBJ3, and we expect that others will be able to build many other interesting systems in much the same way.

In summary, we feel that the OBJ3 system should be useful for the following applications:

1. **Teaching**, especially in the areas of algebraic specification, programming language semantics, and theorem proving [109];

2. **Rapid prototyping**, especially for relatively small but sophisticated systems;

3. **Implementing experimental languages**, especially declarative languages that have features like associative/commutative pattern matching, subtypes, views, theories, and parameterized modules;

4. **Building theorem provers**, for example, an efficient metalogical framework based on equational logic; and

5. **Designing, specifying and documenting** large systems; for example, we used OBJ2 in designing OBJ3, and much of this specification is included in the OBJ3 source code as documentation [100]; see also the recent work on specifying user interface designs via algebraic semiotics [60, 58].

Probably there are other applications that readers have found; please let us know!

A Use of OBJ3

OBJ3's top level statements for declaring objects, modules and views can all be seen as commands whose effect is to add something to the OBJ3 database, which is constructed incrementally. A general "engine" for term rewriting actually does the reductions, consulting the database to get the rules for a given context. Reductions do not change the database in any way (although they may change `memo` tables).

To get started, the only commands you really need are `obj`, `in` and `q` (or `quit`), which get you into OBJ3 from the operating system level, read a file within OBJ3, and quit OBJ3, respectively. The recommended way to work is to first edit a file, then start up OBJ3, and read `in` the file. If you are using an editor like Emacs with buffers that are also UNIX shells, then you can get a log of the execution, containing a record of any problems that arise to help you go back and re-edit the file. If you prefer an editor that does not support shells, then it may be more convenient to use a shell script, and/or to redirect input and output.

As a summary, OBJ3 definitions are assumed to have these properties: the signature should be regular and coherent [76] (coherence means that connected components of sorts must have tops); and the rule set for objects should be Church-Rosser, and if possible, terminating with respect to the given evaluation strategies.

The command `?` (note that there is no ".") produces the following top-level help information

```
Top-level definitional forms include: obj, theory, view, make
The top level commands include:
  q; quit --- to exit from OBJ3
  show .... .  --- for further help: show ? .
  set .... . --- for further help: set ? .
  do .... . --- for further help: do ? .
  apply .....  --- for further help: apply ? .
  other commands:
    in <filename>
    red <term> .
    select <module-expression> .
    cd <directory>; ls; pwd
    start <term> .; show term .
    open [<module-expression>] .; openr [<module-expression>] .; close
    ev <lisp>; evq <lisp>
```

Here ";"s are only used to separate alternatives.

OBJ3 has classes of commands to `show` aspects of the current context, to `set` various system parameters, and to `do` certain actions; in addition, one can evaluate Lisp expressions. Given the argument "?", the commands `show`, `set`, and `do` displays a summary of legal arguments to that command.

The `show` commands are the most numerous. The first group of them have the syntax

`show` ⟨*ModPart*⟩ [⟨*ModExp*⟩]`.`

where ⟨*ModPart*⟩ is one of `sorts`, `psort`, `ops`, `vars`, `eqs`, `mod`, `params`, `subs`, `name`, `sign`, `rules`, `abbrev` and where ⟨*ModExp*⟩ is an optional module expression argument; if there is no module expression, then the module in focus is used. `show` can be abbreviated to `sh`, and `show` ⟨*ModExp*⟩ is short for `show mod` ⟨*ModExp*⟩. The command `select` ⟨*ModExp*⟩ resets the current module to be ⟨*ModExp*⟩, without printing anything. An instance of a ⟨*ModExp*⟩ may also be preceded by `sub` ⟨*Nat*⟩ or `param` ⟨*Nat*⟩ to specify the corresponding sub-module or interface theory, respectively, of the following module; you can even give several of these selectors in a sequence. `show psort` shows the principal sort, `show sign` shows the signature of the selected module, i.e., its sorts and operators; `show params` shows its parameters, if any, `show subs` shows the names of its sub-modules, and `show name` shows its full name. To get more detailed information on a particular sort or operator, you can use the forms

`show sort` ⟨*Sort*⟩ `.`
`show op` ⟨*OpRef*⟩ `.`

which can also be qualified by a module name (see Appendix Section B for details). The command `show all` ⟨*ModExp*⟩ displays the module in a more detailed form. The command `show all rules` [⟨*ModExp*⟩] displays more comprehensively the rules that are used for rewriting in a module. The command `show abbrev` [⟨*ModExp*⟩] shows the abbreviation of the specified module as used when qualifications are being abbreviated).

Warning: If the ⟨*ModExp*⟩ contains a "`.`", you will almost always have to enclose the whole module expression in parentheses.

The command

```
show modules .
```

lists all modules in the current context. The command

```
show time .
```

prints the elapsed time since the last use of this command, or since the start of execution if there has been no prior use. The command

```
show term .
```

shows the current term (the last term reduced, or the last term `started`), while

show [all] rule ⟨*RuleSpec*⟩

shows a specified rule, possibly in detailed form. The command

```
show pending .
```

shows the pending rule applications, while

```
show modes .
```

shows the settings of the settable system parameters, and

```
show all modes .
```

gives more detail. These parameters can all be set with commands of the form

`set` ⟨*Param*⟩ ⟨*Polarity*⟩ `.`

where ⟨*Polarity*⟩ is either `on` or `off`, and ⟨*Param*⟩ is one of `trace`, `blips`, `gc show`, `print with parens`, `show retracts`, `abbrev quals`, `include BOOL`, `clear memo`, `stats`, `trace whole`, `all eqns`, `show var sorts`, `reduce conditions`, or `verbose`. If `trace` is on, then rule applications are displayed during reduction. If `blips` is on, then a "!" is printed whenever a rule is applied, and a "-" is printed whenever an attempt to apply a rule fails. If `gc show` is on, then a message is printed at each start and completion of garbage collection. If `print with parens` is on, then many parentheses are used in printing terms, making the structure much more explicit. If `show retracts` is on, then retracts are shown in the right sides of equations. If `abbrev quals` is on, then qualifications on printed terms or sorts are abbreviated to `MOD`⟨*Nat*⟩. These qualifications are only printed if there are two sorts with the same name or two operators with the same name or pattern and the same arity. If `include BOOL` is on, as it is by default, then `BOOL` is automatically included in modules. If `clear memo` is on, then the memo table is cleared before each reduction. If `stats` is on (which it is by default), then the number of rules applied in each reduction is printed. If `trace whole` is on, then the whole term is printed for each reduction. If `all eqns` is on, then all rule extensions are printed when showing equations. If `show var sorts` is on, then the sorts of variables are printed along with their names, in the form `X:Int`. If the `verbose` mode is on, then modules are displayed in a more detailed form, and id processing will be traced as it is performed.

For commands with the syntax

`do` ⟨*Act*⟩ `.`

⟨*Act*⟩ may be one of `gc` or `clear memo`, or `save` ⟨*ChString*⟩, or `restore` ⟨*ChString*⟩. "`gc`" forces a garbage collection, "`clear memo`" clears the memo table, and `save` and `restore` save the current context, and restore a named context, respectively.

Common Lisp expressions can be evaluated with the syntax

`ev` ⟨*LispExpr*⟩

and the command

`evq` ⟨*LispExpr*⟩

can be used to evaluate a Lisp form with the minimum of output during processing. Longer forms for each of these are `eval` and `eval-quiet`, respectively. These are useful for loading functions used in built-in right sides.

Warning: There are no final periods for these commands, because the Lisp input is considered self-delimiting.

The UNIX interrupt character (typically `^C`) can be used to interrupt OBJ3, which will leave you in the lisp break handler. In Austin Kyoto Common Lisp (AKCL), you can return to the top level of OBJ3 with the break command `:q`, or you can continue the computation with the command `:r`. You can see the current structure of the term being reduced (even if tracing is off) by interrupting OBJ3 (e.g., with `^C`), and then typing `(show)`, followed by a return; then you can resume with `:r`. You can use the command `:h` to get a AKCL break-loop command summary.

In the AKCL version of OBJ3, there are some special command line arguments. These are pairs of these forms

`-in` ⟨*FileName*⟩
`-inq` ⟨*FileName*⟩
`-evq` ⟨*FileName*⟩

The first two forms cause a file to be read in as OBJ3 starts up, either with a trace or quietly. The last form will quietly load a Lisp file on startup.

LaTeX source files containing OBJ examples can be executed using a shell script called `obj3-tex`, by enclosing the OBJ code between matching `\bobj` and `\eobj` commands; everything else is treated as a comment. You will need to have defined the LaTeX commands

```
\newcommand{\bobj}{\begin{alltt}}
\newcommand{\eobj}{\end{alltt}}
```

in your LaTeX source file, and have called the `alltt` style file. A tricky point is that `{` and `}` must be written `\{` and `\}`, respectively, or else they will not be printed by LaTeX. In fact, initial `\`s are stripped off, in these cases, to prepare the text for OBJ execution (this explains why `\{` and `\}` do not appear in`\tt` font in Appendix C.8); this approach extends to allowing arbitrary TeX symbols in OBJ code; for example, `$\ast$` will produce $\ast$. This is very useful for writing papers about OBJ3, which we of course encourage everyone to do; it was used in preparing this paper. The code appears in the distribution tape as `obj/aux/bin/obj3-tex` and the C program is in `obj/aux/c/examples.c`; when `obj3-tex` is run on a file `foo.tex`, it produces a file `foo.obj` containing all the OBJ code that it runs.

Another tricky point is that sometimes code that you want to print relies on some other code that you don't want to print. The shell script saves the day by also executing invisible code, placed between a matching `%\bobj` and `%\eobj` pair, with `obj-tex` stripping off an initial `%` from each line in between.

The distribution tape for Release 2 of OBJ3 comes with all the examples in this paper, in the directory `/obj/exs/`. OBJ3 was implemented on Sun workstations, using Austin Kyoto Common Lisp (AKCL). AKCL has been ported to many different machines, the main requirements being a C compiler and an adequate amount of memory. Release 2.04 of OBJ3 in AKCL has some specific implementation dependent details, but it has been ported to other versions of Common Lisp, including gnu Common Lisp. You can ftp the entire distribution from `ftp://www.cs.ucsd.edu/pub/fac/goguen`. The latest information about the OBJ Family can be obtained from the OBJ homepage at UCSD, at `http://www.cs.ucsd.edu/users/goguen/sys/obj.html`.

B OBJ3 Syntax

This appendix describes the syntax of OBJ3 using the following extended BNF notation: the symbols { and } are used as meta-parentheses; the symbol | is used to separate alternatives; [ ] pairs enclose optional syntax; ... indicates 0 or more repetitions of preceding unit; and "*x*" denotes *x* literally. As an application of this notation, `A`{,`A`}... indicates a non-empty list of `A`'s separated by commas. Finally, `---` indicates comments in the syntactic description, as opposed to comments in OBJ3 code.

`--- top-level ---`

⟨*OBJ-Top*⟩ `::=` {⟨*Object*⟩ | ⟨*Theory*⟩ | ⟨*View*⟩ | ⟨*Make*⟩ | ⟨*Reduction*⟩ |
  `in` ⟨*FileName*⟩ | `quit` | `eof` |

```
    start ⟨Term⟩ . |
    open [⟨ModExp⟩] . | openr [⟨ModExp⟩] . | close |
    ⟨Apply⟩ | ⟨OtherTop⟩}...

⟨Make⟩ ::= make ⟨Interface⟩ is ⟨ModExp⟩ endm

⟨Reduction⟩ ::= reduce [in ⟨ModExp⟩ :] ⟨Term⟩ .

⟨Apply⟩ ::=
   apply {reduction | red | print | retr |
     -retr with sort ⟨Sort⟩ |
     ⟨RuleSpec⟩ [with ⟨VarId⟩ = ⟨Term⟩ {, ⟨VarId⟩ = ⟨Term⟩}... ]}
   {within | at}
   ⟨Selector⟩ {of ⟨Selector⟩}...

⟨RuleSpec⟩ ::= [-][⟨ModId⟩].⟨RuleId⟩
⟨RuleId⟩ ::= ⟨Nat⟩ | ⟨Id⟩

⟨Selector⟩ ::= term | top |
    (⟨Nat⟩...) |
    [ ⟨Nat⟩ [ .. ⟨Nat⟩ ] ] |
    "{" ⟨Nat⟩ {, ⟨Nat⟩}... "}"
  --- note that "()" is a valid selector

⟨OtherTop⟩ ::= ⟨RedLoop⟩ | ⟨Commands⟩ | call-that ⟨Id⟩ . |
    test reduction [in ⟨ModExp⟩ :] ⟨Term⟩ expect: ⟨Term⟩ . | ⟨Misc⟩
  --- "call that ⟨Id⟩ ." is an abbreviation for "let ⟨Id⟩ = ."

⟨RedLoop⟩ ::= rl {. | ⟨ModId⟩} { ⟨Term⟩ .}... .

⟨Commands⟩ ::= cd ⟨Sym⟩ | pwd | ls |
    do ⟨DoOption⟩ . |
    select [⟨ModExp⟩] . |
    set ⟨SetOption⟩ . |
    show [⟨ShowOption⟩] .
  --- in select, can use "open" to refer to the open module

⟨DoOption⟩ ::= clear memo | gc | save ⟨Sym⟩... | restore ⟨Sym⟩... | ?

⟨SetOption⟩ ::= {abbrev quals | all eqns | all rules | blips |
       clear memo | gc show | include BOOL | obj2 | verbose |
       print with parens | reduce conditions | show retracts |
       show var sorts | stats | trace | trace whole}  ⟨Polarity⟩
    | ?

⟨Polarity⟩ ::= on | off

⟨ShowOption⟩ ::=
    {abbrev | all | eqs | mod | name | ops | params | principal-sort |
       [all] rules | select | sign | sorts | subs | vars}
       [⟨ParamSpec⟩ | ⟨SubmodSpec⟩] [⟨ModExp⟩] |
    [all] modes | modules | pending | op ⟨OpRef⟩ | [all] rule ⟨RuleSpec⟩ |
    sort ⟨SortRef⟩ | term | time | verbose | ⟨ModExp⟩ |
    ⟨ParamSpec⟩ | ⟨SubmodSpec⟩ | ?
  --- can use "open" to refer to the open module

⟨ParamSpec⟩ ::= param ⟨Nat⟩
⟨SubmodSpec⟩ ::= sub ⟨Nat⟩
```

```
⟨Misc⟩ ::= eval ⟨Lisp⟩ | eval-quiet ⟨Lisp⟩ | parse ⟨Term⟩ . | ⟨Comment⟩

⟨Comment⟩ ::= *** ⟨Rest-of-line⟩ | ***> ⟨Rest-of-line⟩ |
    *** (⟨Text-with-balanced-parentheses⟩)
⟨Rest-of-line⟩ --- the remaining text of the current line

--- modules ---

⟨Object⟩ ::= obj ⟨Interface⟩ is {⟨ModElt⟩ | ⟨Builtins⟩}... endo

⟨Theory⟩ ::= th ⟨Interface⟩ is ⟨ModElt⟩... endth

⟨Interface⟩ ::= ⟨ModId⟩ [[⟨ModId⟩... :: ⟨ModExp⟩ {, ⟨ModId⟩... :: ⟨ModExp⟩}... ]]

⟨ModElt⟩ ::=
    {protecting | extending | including | using} ⟨ModExp⟩ . |
    using ⟨ModExp⟩ with ⟨ModExp⟩ {and ⟨ModExp⟩}... |
    define ⟨SortId⟩ is ⟨ModExp⟩ . |
    principal-sort ⟨Sort⟩ . |
    sort ⟨SortId⟩... . |
    subsort ⟨Sort⟩... { < ⟨Sort⟩... }... . |
    as ⟨Sort⟩ : ⟨Term⟩ if ⟨Term⟩ . |
    op ⟨OpForm⟩ : ⟨Sort⟩... -> ⟨Sort⟩ [⟨Attr⟩] . |
    ops {⟨Sym⟩ | (⟨OpForm⟩)}... : ⟨Sort⟩... -> ⟨Sort⟩ [⟨Attr⟩] . |
    op-as ⟨OpForm⟩ : ⟨Sort⟩... -> ⟨Sort⟩ for ⟨Term⟩ if ⟨Term⟩ [⟨Attr⟩] . |
    [⟨RuleLabel⟩] let ⟨Sym⟩ [: ⟨Sort⟩] = ⟨Term⟩ . |
    var ⟨VarId⟩... : ⟨Sort⟩ . |
    vars-of [⟨ModExp⟩] . |
    [⟨RuleLabel⟩] eq ⟨Term⟩ = ⟨Term⟩ . |
    [⟨RuleLabel⟩] cq ⟨Term⟩ = ⟨Term⟩ if ⟨Term⟩ . |
    ⟨Misc⟩

⟨Attr⟩ ::= [ {assoc | comm | {id: | idr:} ⟨Term⟩ | idem | memo |
    strat (⟨Int⟩... ) | prec ⟨Nat⟩ | gather ({e | E | &}... ) |
    poly ⟨Lisp⟩ | intrinsic}... ]

⟨RuleLabel⟩ ::= ⟨Id⟩... {, ⟨Id⟩... }...

⟨ModId⟩ --- simple identifier, by convention all caps
⟨SortId⟩ --- simple identifier, by convention capitalized
⟨VarId⟩ --- simple identifier, typically capitalized
⟨OpName⟩ ::= ⟨Sym⟩ {"_" | " " | ⟨Sym⟩}...
⟨Sym⟩ --- any operator syntax symbol (blank delimited)
⟨OpForm⟩ ::= ⟨OpName⟩ | (⟨OpName⟩)
⟨Sort⟩ ::= ⟨SortId⟩ | ⟨SortId⟩.⟨SortQual⟩
⟨SortQual⟩ ::= ⟨ModId⟩ | (⟨ModExp⟩)
⟨Lisp⟩ --- a Lisp expression
⟨Nat⟩ --- a natural number
⟨Int⟩ --- an integer

⟨Builtins⟩ ::=
    bsort ⟨SortId⟩ ⟨Lisp⟩ . |
    [⟨RuleLabel⟩] bq ⟨Term⟩ = ⟨Lisp⟩ . |
    [⟨RuleLabel⟩] beq ⟨Term⟩ = ⟨Lisp⟩ . |
    [⟨RuleLabel⟩] cbeq ⟨Term⟩ = ⟨Lisp⟩ if ⟨BoolTerm⟩ . |
    [⟨RuleLabel⟩] cbq ⟨Term⟩ = ⟨Lisp⟩ if ⟨BoolTerm⟩ .
```

```
--- views ---

⟨View⟩ ::= view [⟨ModId⟩] from ⟨ModExp⟩ to ⟨ModExp⟩ is ⟨ViewElt⟩... endv |
    view ⟨ModId⟩ of ⟨ModExp⟩ as ⟨ModExp⟩ is ⟨ViewElt⟩... endv

--- terms ---

⟨Term⟩ ::= ⟨Mixfix⟩ | ⟨VarId⟩ | (⟨Term⟩) |
    ⟨OpName⟩ (⟨Term⟩ {, ⟨Term⟩}... ) | (⟨Term⟩).⟨OpQual⟩
  --- precedence and gathering rules used to eliminate ambiguity

⟨OpQual⟩ ::= ⟨Sort⟩ | ⟨ModId⟩ | (⟨ModExp⟩)
⟨Mixfix⟩ --- mixfix operator applied to arguments

--- module expressions ---

⟨ModExp⟩ ::= ⟨ModId⟩ | ⟨ModId⟩ is ⟨ModExpRenm⟩ | ⟨ModExpRenm⟩ + ⟨ModExp⟩ |
    ⟨ModExpRenm⟩

⟨ModExpRenm⟩ ::= ⟨ModExpInst⟩ * (⟨RenameElt⟩ {, ⟨RenameElt⟩}... ) | ⟨ModExpInst⟩

⟨ModExpInst⟩ ::= ⟨ParamModExp⟩[⟨Arg⟩ {,⟨Arg⟩}... ] | (⟨ModExp⟩)

⟨ParamModExp⟩ ::= ⟨ModId⟩ | (⟨ModId⟩ * (⟨RenameElt⟩ {, ⟨RenameElt⟩}... ))

⟨RenameElt⟩ ::= sort ⟨SortRef⟩ to ⟨SortId⟩ | op ⟨OpRef⟩ to ⟨OpForm⟩

⟨Arg⟩ ::= ⟨ViewArg⟩ | ⟨ModExp⟩ | [sort] ⟨SortRef⟩ | [op] ⟨OpRef⟩
--- may need to precede ⟨SortRef⟩ by "sort" and ⟨OpRef⟩ by "op" to
--- distinguish from general case (i.e., from a module name)

⟨ViewArg⟩ ::= view [from ⟨ModExp⟩] to ⟨ModExp⟩ is ⟨ViewElt⟩... endv

⟨ViewElt⟩ ::= sort ⟨SortRef⟩ to ⟨SortRef⟩ . | var ⟨VarId⟩... : ⟨Sort⟩ . |
    op ⟨OpExpr⟩ to ⟨Term⟩ . | op ⟨OpRef⟩ to ⟨OpRef⟩ .
  --- priority given to ⟨OpExpr⟩ case
  --- vars are declared with sorts from source of view (a theory)

⟨SortRef⟩ ::= ⟨Sort⟩ | (⟨Sort⟩)
⟨OpRef⟩ ::= ⟨OpSpec⟩ | (⟨OpSpec⟩) | (⟨OpSpec⟩).⟨OpQual⟩ | ((⟨OpSpec⟩).⟨OpQual⟩)
  --- in views if have (op).(M) must be enclosed in (), i.e., ((op).(M))
⟨OpSpec⟩ ::= ⟨OpName⟩ | ⟨OpName⟩ : ⟨SortId⟩... -> ⟨SortId⟩
⟨OpExpr⟩  --- a ⟨Term⟩ consisting of a single operator applied to variables

--- equivalent forms ---

assoc = associative         comm = commutative
cq = ceq                    dfn = define
ev = eval                   evq = eval-quiet
jbo = endo                  ht = endth
endv = weiv = endview       ex = extending
gather = gathering          id: = identity:
idem = idempotent           idr: = identity-rules:
in = input                  inc = including
obj = object                poly = polymorphic
prec = precedence           psort = principal-sort
pr = protecting             q = quit
red = reduce                rl = red-loop
```

```
sh = show                    sorts = sort
strat = strategy             subsorts = subsort
th = theory                  us = using
vars = var                   *** = ---
***> = --->

--- Lexical analysis ---

--- Tokens are sequences of characters delimited by blanks
--- "(", ")", and "," are always treated as single character symbols
--- Tabs and returns are equivalent to blanks (except inside comments)
--- Normally, "[", "]", "_", ",", "{", and "}"
--- are also treated as single character symbols.
```

C More Examples

This appendix contains a number of examples that illustrate the power and flexibility of OBJ's unusual features, including hierarchical parameterized modules, subsorts, and rewriting modulo attributes.

C.1 Some Set Theory

The following two objects define some constructions on sets that may be useful in other examples, such as the category theory example in Appendix C.8. The use of `memo` has quite a significant effect on the test cases.

Warning: These particular definitions are very inefficient. The example `set2.obj` distributed with OBJ3 provides a more efficient alternative.

```
obj BSET[X :: TRIV] is
  sort Set .
  pr IDENTICAL .
  ops ({}) omega : -> Set .
  op {_} : Elt -> Set .
  op _+_ : Set Set -> Set [assoc comm id: ({})] .        *** exclusive or
  op _&_ : Set Set -> Set [assoc comm idem id: omega] .  *** intersection
  vars S S' S'' : Set .   vars E E' : Elt .
  eq S + S = {} .
  cq { E } & { E' } = {} if E =/= E' .
  eq S & {} = {} .
  cq S & (S' + S'') = (S & S') + (S & S'')
      if (S' =/== {}) and (S'' =/== {}) .
    *** made conditional as an example of how to avoid non-termination
    *** from identity completion (in fact, not required)
endo

obj SET[X :: TRIV] is
  protecting BSET[X] .
  protecting INT .
  op _U_ : Set Set -> Set [assoc comm id: ({})] .
  op _-_ : Set Set -> Set .
  op #_ : Set -> Int [prec 0] .
  op _in_ : Elt Set -> Bool .
  op _in_ : Set Set -> Bool .
  op empty?_ : Set -> Bool .
  var X : Elt .   vars S S' S'' : Set .
  eq S U S' = (S & S') + S + S' .
  eq S - S' = S + (S & S') .
  eq empty? S = S == {} .
```

```
  eq X in S = { X } & S =/= {} .
  eq S in S' = S U S' == S' .
  eq # {} = 0 .
  cq #({ X } + S) = # S if X in S .
  cq #({ X } + S) = 1 + # S if not X in S .
endo

*** test cases
obj SET-OF-INT is
  protecting SET[INT] .
  ops s1 s2 s3 : -> Set [memo] .
  eq s1 = { 1 } .
  eq s2 = s1 U { 2 } .
  eq s3 = s2 U { 3 } .
endo
reduce s3 .               ***> should be: {1,2,3}
reduce # s3 .             ***> should be: 3
reduce (s2 U s1) .        ***> should be: {1,2}
reduce #(s3 U s1) .       ***> should be: 3
reduce empty?(s3 + s3) .  ***> should be: true
reduce empty?(s1 + s3) .  ***> should be: false
reduce 3 in s2 .          ***> should be: false
reduce s1 in s3 .         ***> should be: true
reduce s1 - s3 .          ***> should be: {}
reduce s3 - s2 .          ***> should be: {3}
reduce s3 & s1 .          ***> should be: {1}
reduce s3 & s2 .          ***> should be: {1,2}
reduce omega U s2 .       ***> should be: omega
```

C.2 A Simple Programming Language

It is generally rather straightforward to write specifications of programming languages in OBJ, as we hope the following example shows. (This example has been adapted from [69]; the first such use of OBJ occurs in [79].) This example is preceded by two standard generic modules, which are also used in some other examples below.

```
obj LIST[X :: TRIV] is
  sorts List NeList .
  op nil : -> List .
  subsorts Elt < NeList < List .
  op __ : List List -> List [assoc id: nil] .
  op __ : NeList List -> NeList .
  op __ : NeList NeList -> NeList .
  protecting NAT .
  op |_| : List -> Nat .
  eq | nil | = 0 .
  var E : Elt .   var L : List .
  eq | E L | = 1 + | L | .
  op tail_ : NeList -> List [prec 120] .
  var E : Elt .   var L : List .
  eq tail E L = L .
endo

obj ARRAY[INDEX VALUE :: TRIV] is
  sort Array .
  op nilArr : -> Array .
  op put : Elt.INDEX Elt.VALUE Array -> Array .
  op _[_] : Array Elt.INDEX -> Elt.VALUE .
  op _in_ : Elt.INDEX Array -> Bool .
```

```
  op undef : Elt.INDEX -> Elt.VALUE .  *** err-op
  var A : Array .
  var I I' : Elt.INDEX .   var E : Elt.VALUE .
  eq put(I,E,A)[ I ] = E .
  ceq put(I,E,A)[ I' ] = A [ I' ] if I =/= I' .
  eq I in nilArr = false .
  eq I in put(I',E,A) = I == I' or I in A .
  ceq A [ I ] = undef(I) if not I in A .  *** err-eqn
endo

*** the expressions of Fun
obj EXP is
  dfn Env is ARRAY[QID,INT] .
  sorts IntExp BoolExp .
  subsorts Int Id < IntExp .
  subsorts Bool < BoolExp .
  ops (_and_) (_or_) : BoolExp BoolExp -> BoolExp .
  op not_ : BoolExp -> BoolExp .
  op _<_ : IntExp IntExp -> BoolExp .
  op _=_ : IntExp IntExp -> BoolExp .
  op if_then_else_fi : BoolExp IntExp IntExp -> IntExp .
  ops (_+_) (_-_) (_*_) : IntExp IntExp -> IntExp .
  op [[_]]_ : IntExp Env -> Int .
  op [[_]]_ : BoolExp Env -> Bool .
  var N : Int .         var T : Bool .
  vars E E' : IntExp .  vars B B' : BoolExp .
  var I : Id .          var V : Env .
  eq [[ N ]] V = N .
  eq [[ I ]] V  = V [ I ] .
  eq [[ E + E' ]] V = ([[ E ]] V) + ([[ E' ]] V) .
  eq [[ E - E' ]] V = ([[ E ]] V) - ([[ E' ]] V) .
  eq [[ E * E' ]] V = ([[ E ]] V) * ([[ E' ]] V) .
  eq [[ T ]] V = T .
  eq [[ E < E' ]] V = ([[ E ]] V) < ([[ E' ]] V) .
  eq [[(E = E')]] V = ([[ E ]] V) == ([[ E' ]] V) .
  eq [[ B and B' ]] V = ([[ B ]] V) and ([[ B' ]] V) .
  eq [[ B or  B' ]] V = ([[ B ]] V) or  ([[ B' ]] V) .
  eq [[ not B ]] V = not([[ B ]] V) .
  eq [[ if B then E else E' fi ]] V =
        if [[ B ]] V then [[ E ]] V else [[ E' ]] V fi .
endo

*** the statements of Fun
obj STMT is
  sort Stmt .
  protecting EXP .
  op _;_ : Stmt Stmt -> Stmt [assoc] .
  op _:=_ : Id IntExp -> Stmt .
  op while_do_od : BoolExp Stmt -> Stmt .
  op [[_]]_ : Stmt Env -> Env .
  vars S S' : Stmt .    var V : Env .
  var E : IntExp .      var B : BoolExp .
  var I : Id .
  eq [[ I := E ]] V = put(I,[[ E ]] V, V) .
  eq [[ S ; S' ]] V = [[ S' ]] [[ S ]] V .
  eq [[ while B do S od ]] V = if [[ B ]] V then
     [[ while B do S od ]] [[ S ]] V else V fi .
endo
```

```
*** evaluation of Fun programs
obj FUN is
  sorts Fun Init .
  protecting STMT .
  dfn IdList is LIST[QID] .
  dfn IntList is LIST[INT] .
  dfn InitList is (LIST *(op nil to nil-init, op (__) to (_;_)))[Init] .
  op _initially_ : Id IntExp -> Init [prec 1].
  op fun _ _ is vars _ body: _ : Id IdList InitList Stmt -> Fun .
  op [[_:=_]]_ : IdList IntList Env -> Env .
  op [[_]]_ : InitList Env -> Env .
  op [[_]][_]_ : Fun Env IntList -> Env .
  op [[_]]_ : Fun IntList -> Int .
  op wrong#args : -> Env .   *** err-op
  vars I F : Id .      var Is : IdList .
  var N : Int .        var Ns : IntList .
  var E : IntExp .     var INs : InitList .
  var S : Stmt .       var V : Env .
  eq [[ nil-init ]] V = V .
  eq [[(I initially E); INs ]] V = [[ INs ]] [[ I := E ]] V .
  eq [[ I Is := N Ns ]] V = ([[ I := N ]] ([[ Is := Ns ]] V)).STMT .
  eq [[(nil).IdList := (nil).IntList ]] V = V .
  eq [[ fun F(Is) is vars nil-init body: S ]][ V ](Ns) = [[ S ]] V .
  eq [[ fun F(Is) is vars INs body: S ]][ V ](Ns) =
        [[ S ]] [[ INs ]] [[ Is := Ns ]] V .
  eq [[ fun F(Is) is vars INs body: S ]](Ns) =
        [[ fun F(Is) is vars INs body: S ]][ nilArr ](Ns) [ F ] .
  cq [[ Is := Ns ]] V = wrong#args if | Is | =/= | Ns | .  *** err-qn
endo

*** pow(n m) finds the nth power of m for positive n or 0
reduce [[ fun 'pow('n 'm) is vars 'pow initially 1 body:
          while 0 < 'n do ('pow := 'pow * 'm);('n := 'n - 1) od ]](4 2) .
***> should be: 16

*** factorial of n
reduce [[ fun 'fac('n) is vars ('fac initially 1);('i initially 0) body:
          while 'i < 'n do ('i := 'i + 1); ('fac := 'i * 'fac) od ]](5) .
***> should be: 120

*** max finds the maximum of a list of three numbers
reduce [[ fun 'max('a 'b 'c) is vars 'n initially 2 body:
          ('max := 'a); while 0 < 'n do
          ('n := 'n - 1); ('x := 'b); ('b := 'c);
          ('max := if 'x < 'max then 'max else 'x fi) od ]](3 123 32) .
***> should be: 123
```

C.3 Unification

The use of associative and/or commutative matching allows writing a simple and elegant unification algorithm in OBJ3. A more efficient version of this algorithm can be used to implement logic (i.e., relational) programming on the Rewrite Rule Machine; see [75, 107] for more detail.

In this code, a term is either a variable (such as `'X`) or else is of the form `F[T]`, where `F` is an operator symbol (such as `'F`) and `T` is a list of terms; a constant is of the form `F[nil]`. An equation is a pair of terms separated by `=`, and a system of equations is a list of equations, separated by `&` signs. The symbols { and } are used to delimit subsystems of equations. The form `let X be T1 in T2` indicates the substitution of `T1` for `X` in `T2`; this operator extends to term lists, equations and systems. A system to be unified, presented in the form {{ `S` }}, is reduced to another system { `S'` } in "solved form", by progressively moving solved equations out of the inner brackets, so that in {S1 & {S2}}, the equations in S1 are solved, while those in S2 are not.

```
obj SUBST is
  sorts Eqn Term .
  protecting QID .
  subsort Id < Term .
  pr TERMS is (LIST *(sort List to TermList, sort NeList to NeTermList))[Term].
  dfn Op is QID .
  op _[_] : Op TermList -> Term [prec 1] .
  op _=_ : Term Term -> Eqn [comm prec 120] .
  pr SYSTEM is (LIST *(sort List to System, sort NeList to NeSystem,
     op nil to null, op (__) to (_&_)))[Eqn].
  op {_} : System -> System .   *** scope delimiter
  op _=_ : TermList TermList -> System [comm prec 120] .
  vars T U V : Term .   var Us : NeTermList .
  var S : NeSystem .    var Ts : TermList .
  eq (T Ts = U Us) = (T = U) & (Ts = Us).
  op let_be_in_ : Id Term Term -> Term .
  op let_be_in_ : Id Term TermList -> TermList .
  op let_be_in_ : Id Term Eqn -> Eqn .
  op let_be_in_ : Id Term System -> System .
  vars X Y : Id .     var F : Op .
  eq let X be T in nil = nil .
  eq let X be T in Y = if X == Y then T else Y fi .
  eq let X be T in F[Ts] = F[let X be T in Ts].
  eq let X be T in (U Us) = (let X be T in U) (let X be T in Us).
  eq let X be T in (U = V) = (let X be T in U) = (let X be T in V) .
  eq let X be T in null = null .
  eq let X be T in ((U = V) & S) = (let X be T in (U = V)) & (let X be T in S).
endo

***> first without occur check
obj UNIFY is
  using SUBST with SYSTEM and TERMS .
  op unify_ : System -> System [prec 120].
  op fail : -> Eqn .
  var T : Term .            vars Ts Us : NeTermList .
  vars S S' S'' : System .  var X : Id .
  eq unify S = {{S}} .
  eq S & (T = T) & S' = S & S' .
  eq S & fail & S' = fail .
  eq let X be T in fail = fail .
  eq {null} = null .
  eq {fail} = fail .
  vars F G : Op .   vars X : Id .
  eq {S & (F[Ts] = G[Us]) & S'} = if F == G and | Ts | == | Us |
     then {S & (Ts = Us) & S'} else fail fi .
  eq {S & {S' & (X = T) & S''}} = if X == T then {S & {S' & S''}} else
     {(X = T) & (let X be T in S) & {let X be T in S' & S''}} fi .
endo
reduce unify 'f['g['X] 'Y] = 'f['g['h['Y]] 'h['Z]].
reduce unify 'f['X 'Y] = 'f['Y 'g['Y]].
reduce unify ('f['g['X] 'Y] = 'f['g['h['Y]] 'h['Z]]) & ('h['X] = 'Z).
reduce unify 'f['X 'g['Y]] = 'f['Z 'Z].
reduce unify 'f['X 'g['Y]] = 'f['Z].
reduce unify 'f['Y 'g['Y]] = 'f['h['Z] 'Z].
reduce unify 'f['Y 'a[nil]] = 'f['g['a[nil]] 'Z].
```

```
***> now add occur check
obj UNIFY-OCH is using UNIFY .
  op _in_ : Id TermList -> Bool .
  vars X Y : Id .   var F : Op .
  var T : Term .    var Ts : NeTermList .
  eq X in Y = X == Y .
  eq X in F[Ts] = X in Ts .
  eq X in T Ts = X in T or X in Ts .
  cq (X = T) = fail if X in T .
endo
reduce unify 'f['g['X] 'Y] = 'f['g['h['Y]] 'h['Z]].
reduce unify 'f['X 'Y] = 'f['Y 'g['Y]].
reduce unify ('f['g['X] 'Y] = 'f['g['h['Y]] 'h['Z]]) & ('h['X] = 'Z).
reduce unify 'f['X 'g['Y]] = 'f['Z 'Z].
reduce unify 'f['X 'g['Y]] = 'f['Z].
reduce unify 'f['Y 'g['Y]] = 'f['h['Z] 'Z].
reduce unify 'f['Y 'a[nil]] = 'f['g['a[nil]] 'Z].
```

C.4 Some Theorem Proving

Because OBJ3 is rigorously based upon order sorted equational logic, every OBJ3 computation proves some theorem. By choosing the right specification and the right term, these computations can be made to prove interesting theorems, as the following examples demonstrate. It is not enough just to give the OBJ code for a proof – called its *proof score* – and then do the computation; it is also necessary to show that if all reductions in the proof score evaluate to `true`, then the theorem really has been proved. Many such justifications are given in [51], from which the proof scores in this section were taken. Further justifications, and some more complex proofs, including the verification of parameterized modules, and some hardware circuits, may be found in [59]; see also [66]. Such proof scores are generated and checked by Kumo [65, 64, 78, 60].

The following simple specification of the natural numbers is used in several examples below.

```
obj NAT is
  sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat [prec 1] .
  op _+_ : Nat Nat -> Nat .
  vars L M N : Nat .
  eq M + 0 = M .
  eq M + s N = s(M + N) .
endo
```

C.4.1 Associativity of Addition

The following proves that addition of natural numbers is associative.

```
open .
ops l m n : -> Nat .

*** base case, n=0: l+(m+0)=(l+m)+0
reduce l + (m + 0) == (l + m) + 0 .

*** induction step
eq l + (m + n) = (l + m) + n .
reduce l + (m + s n) == (l + m) + s n .
close

*** thus we can assert
obj ASSOC is
  protecting NAT .
  vars-of NAT .
```

```
  eq L + (M + N) = (L + M) + N .
endo
```

C.4.2 Commutativity of Addition

The following proves that addition of natural numbers is commutative.

```
open .
vars-of NAT .
ops m n : -> Nat .

*** show lemma0: 0 + n = n, by induction on n
*** base for lemma0, n=0
reduce 0 + 0 == 0 .
*** induction step
eq 0 + n = n .
reduce 0 + (s n) == s n .
*** thus we can assert
eq 0 + N = N .

*** show lemma1: (s m) + n = s(m + n), again by induction on n
*** base for lemma1, n=0
reduce (s m)+ 0 == s(m + 0) .
*** induction step
eq (s m)+ n = s(m + n) .
reduce (s m) + (s n) == s(m + s n) .
*** thus we can assert
eq (s M)+ N = s(M + N).

*** show m + n = n + m, again by induction on n
*** the base case, n=0
reduce m + 0 == 0 + m .
*** induction step
eq m + n = n + m .
reduce m + (s n) == (s n) + m .
close
```

Of course, we must not assert commutativity as a rewrite rule, or we would get non-terminating reductions. The above two proofs show that we are entitled to use associative-commutative rewriting for `+`, and we do so below.

It is interesting to contrast the above proofs with corresponding proofs due to Paulson in Cambridge LCF [129]. The LCF proofs are much more complex, in part because LCF functions are partial, and therefore must be proved total, whereas functions are automatically total (on their domain) in equational logic.

C.4.3 Formula for $1 + ... + n$

We now give a standard inductive proof over the natural numbers, the formula for the sum of the first n positive natural numbers,

$1 + 2 + \ldots + n = n(n + 1)/2.$

Here we take advantage of the two results proven above by giving `+` the attributes `assoc` and `comm`; the score, as given, will not work if either (or both) of these attributes are deleted. This application of associative/commutative rewriting saves the user from having to worry about the ordering and grouping of subterms within terms headed by `+`.

```
obj NAT is
  sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat [prec 1] .
  op _+_ : Nat Nat -> Nat [assoc comm] .
  vars M N : Nat .
```

```
  eq M + 0 = M .
  eq M + s N = s(M + N).
  op _*_ : Nat Nat -> Nat .
  eq M * 0 = 0 .
  eq M * s N = (M * N)+ M .
endo

open .
vars-of NAT .
ops m n : -> Nat .

*** first show two lemmas, 0*n=0 and (sm)*n=(m*n)+n
*** base for first lemma
reduce 0 * 0 == 0 .
*** induction step for first lemma
eq 0 * n = 0 .
reduce 0 * s n == 0 .
*** thus we can assert
eq 0 * N = 0 .

*** base for second lemma
reduce (s n)* 0 == (n * 0) + 0 .
*** induction step for second lemma
eq (s m) * n = (m * n)+ n .
reduce (s m)*(s n) == (m * s n)+ s n .
*** thus
eq (s M)* N = (M * N)+ N .

*** now define
op sum : Nat -> Nat .
eq sum(0) = 0 .
eq sum(s N) = (s N)+ sum(N) .

*** show sum(n)+sum(n)=n*sn
*** base case
reduce sum(0) + sum(0) == 0 * s 0 .
*** induction step
eq sum(n) + sum(n) = n * s n .
reduce sum(s n) + sum(s n) == (s n)*(s s n) .
close
```

C.4.4 Fermat's Little Theorem for $p = 3$

The so-called "little Fermat theorem" says that

$x^p \equiv x \pmod{p}$

for any prime p, i.e., that the remainder of x^p by p equals the remainder of x by p. The following OBJ3 proof score for the case $p = 3$ needs a slightly more sophisticated natural number object which assumes that we have already proven that multiplication is associative and commutative. This is a nice example of an inductive proof where there are non-trivial relations among the constructors. (We thank Dr. Emmanuel Kounalis for doubting that OBJ3 could handle non-trivial relations on constructors, and then presenting the challenge to prove this result.)

```
obj NAT is
  sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat [prec 1] .
  op _+_ : Nat Nat -> Nat [assoc comm] .
  vars L M N : Nat .
  eq M + 0 = M .
```

```
  eq M + s N = s(M + N) .
  op _*_ : Nat Nat -> Nat [assoc comm] .
  eq M * 0 = 0 .
  eq M * s N = (M * N)+ M .
  eq L * (M + N) = (L * M) + (L * N) .
  eq M + M + M = 0 .
endo

*** base case, x = 0
reduce 0 * 0 * 0 == 0 .
*** induction step
open .
op x : -> Nat .
eq x * x * x = x .
reduce (s x)*(s x)*(s x) == s x .
close
```

The same technique can be used for $p = 5$, $p = 7$, etc., but something more sophisticated is needed to get the result for all primes.

C.4.5 Left and Right Group Axioms

A standard example in group theory is to prove that the right handed versions of the axioms follow from the left handed versions. It is straightforward to do this example using OBJ's `apply` feature. The terms following the numbers in square brackets (e.g., `[0]`) show what the result should be.

```
th GROUPL is
  sort Elt .
  op _*_ : Elt Elt -> Elt .
  op e : -> Elt .
  op _-1 : Elt -> Elt [prec 2] .
  var A B C : Elt .
  [lid] eq e * A = A .
  [lnv] eq A -1 * A = e .
  [las] eq A * (B * C) = (A * B) * C .
endth

open .
op a : -> Elt .

*** first, prove the right inverse law:
start a * a -1 .
 ***> [0] (a * a -1)
apply -.lid at term .
 ***> [1] e * (a * a -1)
apply -.lnv with A = (a -1) within term .
 ***> [2] ((a -1) -1 * a -1) * (a * a -1)
apply .las at term .
 ***> [3] ((a -1 -1 * a -1)* a)* a -1
apply -.las with A = (a -1 -1) within term .
 ***> [4] ((a -1 -1 * (a -1 * a)) * a -1
apply .lnv within term .
 ***> [5] (a -1 -1 * e) * a -1
apply -.las at term .
 ***> [6] a -1 -1 * (e * a -1)
apply .lid within term .
 ***> [7] a -1 -1 * a -1
apply .lnv at term .
 ***> [8] e
```

```
*** we can now add the proven equation
 [rnv] eq (A * (A -1)) = e .

*** next, we prove the right identity law:
start a * e .
 ***> [0] a * e
apply -.lnv with A = a within term .
 ***> [1] a *(a -1 * a)
apply .las at term .
 ***> [2] (a * a -1)* a
apply .rnv within term .
 ***> [3] e * a
apply .lid at term .
 ***> [4] a

***> we can add the proven equation
 [rid] eq A * e = A .
close
```

This example can be simplified by assuming associativity of the group multiplication as an attribute:

```
th GROUPLA is
  sort Elt .
  op _*_ : Elt Elt -> Elt [assoc] .
  op e : -> Elt .
  op _-1 : Elt -> Elt [prec 2] .
  var A : Elt .
  [lid]  eq e * A = A .
  [linv] eq A -1 * A = e .
endth

open .
op a : -> Elt .

*** first, prove the right inverse law:
start a * a -1 .
apply -.lid at term .
  ***> should be: e * a * a -1
apply -.linv with A = (a -1) within term .
  ***> should be: a -1 -1 * a -1 * a * a -1
apply .linv at [2 .. 3] of term .
  ***> should be: a -1 -1 * e * a -1
apply reduction at term .
  ***> should be: e

*** add the proven equation:
 [rinv] eq A * A -1 = e .

*** second prove the right identity law:
start a * e .
apply -.linv with A = a within term .
  ***> should be: a * a -1 * a
apply .rinv at [1 .. 2] .
  ***> should be: e * a
apply reduction at term .
  ***> should be: a

*** add the proven equation:
```

```
 [rid] eq A * e = A .
close
```

C.4.6 Injective Functions

Proving that an injective function with a right inverse is an isomorphism gives a good illustration of `apply` when there are conditional equations.

```
th INJ is
  sorts A B .
  op f_ : A -> B .
  op g_ : B -> A .
  var A : A . vars B B' : B .
  [lnv] eq g f A = A .
  [inj] cq B = B' if g B == g B' .
endth

open .
op b : -> B .

start f g b .
apply .inj with B' = b at term .
apply red at term .
***> should be: b
close
```

What happens here is that, in order to apply the rule `.inj` to `f g b` with `B' = b`, we must first prove that the condition is true, which in this case is that `g f g b == g b`. Therefore, OBJ3 shifts its focus from the original term for reduction, to the condition, so that `red` (i.e., reduction) is actually applied to `g f g b == g b`. In fact, the rule `.lnv` applies, to give `g b == g b`, which reduces to `true` by a built in rule for `==`. Therefore the given term, `f g b` is rewritten to `b`. This establishes the equation

```
eq f g b = b .
```

and hence that

```
eq f g b = b .
```

so that `g` is indeed as isomorphism.

C.5 Lazy Evaluation

This subsection gives the famous Sieve of Erasthothenes, which finds all the prime numbers. Since this is an infinite structure, laziness is needed to actually run it.

```
obj LAZYLIST[X :: TRIV] is
  sort List .
  subsort Elt < List .
  op nil : -> List .
  op __ : List List -> List [assoc idr: nil strat (0)] .
endo

obj SIEVE is
  protecting LAZYLIST[INT] .
  op force : List List -> List [strat (1 2 0)] .
  op show_upto_ : List Int -> List .
  op filter_with_ : List Int -> List .
  op ints-from_ : Int -> List .
  op sieve_ : List -> List .
  op primes : -> List .
  var P I E : Int .
```

```
  var S L : List .
  eq force(L,S) = L S .
  eq show nil upto I = nil .
  eq show E S upto I = if I == 0 then nil
     else force(E,show S upto (I - 1)) fi .
  eq filter nil with P = nil .
  eq filter I S with P = if (I rem P) == 0 then filter S with P
     else I (filter S with P) fi .
  eq ints-from I = I (ints-from (I + 1)) .
  eq sieve nil = nil .
  eq sieve (I S) = I (sieve (filter S with I)) .
  eq primes = sieve (ints-from 2) .
endo

reduce show primes upto 10 .
***> should be: 2 3 5 7 11 13 17 19 23 29
```

C.6 Combinators

The convention for terms in combinatory algebra requires the use of gathering attributes. Some rather nice calculations can then be done, in exactly the usual notation. Here is the basic object:

```
obj COMBINATORS is
  sort T .
  op __ : T T -> T [gather (E e)]. *** forces left association
  ops S K I : -> T .
  vars M N P : T .
  eq K M N = M .
  eq I M = M .
  eq S M N P = (M P) (N P).
endo
```

Now the reductions, all of which should evaluate to `true`, because all of them correspond to identities of combinatory algebra:

```
open .
ops m n p : -> T .

red S K K m == I m .
red S K S m == I m .
red S I I I m == I m .

red K m n == S(S(K S)(S(K K)K))(K(S K K))m n .
red S m n p == S(S(K S)(S(K(S(K S)))(S(K(S(K K)))S)))(K(K(S K K)))m n p .
red S(K K) m n p == S(S(K S)(S(K K)(S(K S)K)))(K K) m n p .

let X = S I .
red X X X X m == X(X(X X)) m .

close
```

The last of these takes 27 rewrites, and is not the sort of thing that one would like to do by hand!

C.7 A Number Hierarchy

The various number systems used in modern mathematics exhibit a very rich hierarchy of sorts and subsorts, from the nonzero natural numbers up to the quarternions. The way this example avoids division by zero is also a nice illustration of using order sorted algebra to define functions on subsorts. (Most of the work on this example was done by Dr. José Meseguer.)

```
obj NAT is
  sorts Nat NzNat Zero .
  subsorts Zero NzNat < Nat .
  op 0 : -> Zero .
  op s_ : Nat -> NzNat .
  op p_ : NzNat -> Nat .
  op _+_ : Nat Nat -> Nat [assoc comm] .
  op _*_ : Nat Nat -> Nat .
  op _*_ : NzNat NzNat -> NzNat .
  op _>_ : Nat Nat -> Bool .
  op d : Nat Nat -> Nat [comm] .
  op quot : Nat NzNat -> Nat .
  op gcd : NzNat NzNat -> NzNat [comm] .
  vars N M : Nat .   vars N' M' : NzNat .
  eq p s N = N .
  eq N + 0 = N .
  eq (s N) + (s M) = s s (N + M) .
  eq N * 0 = 0 .
  eq 0 * N = 0 .
  eq (s N) * (s M) = s (N + (M + (N * M))) .
  eq 0 > M = false .
  eq N' > 0 = true .
  eq s N > s M = N > M .
  eq d(0,N) = N .
  eq d(s N, s M) = d(N,M) .
  eq quot(N,M') = if ((N > M') or (N == M')) then s quot(d(N,M'),M')
     else 0 fi .
  eq gcd(N',M') = if N' == M' then N' else (if N' > M' then
     gcd(d(N',M'),M') else gcd(N',d(N',M')) fi) fi .
endo

obj INT is
  sorts Int NzInt .
  protecting NAT .
  subsort Nat < Int .
  subsorts NzNat < NzInt < Int .
  op -_ : Int -> Int .
  op -_ : NzInt -> NzInt .
  op _+_ : Int Int -> Int [assoc comm] .
  op _*_ : Int Int -> Int .
  op _*_ : NzInt NzInt -> NzInt .
  op quot : Int NzInt -> Int .
  op gcd : NzInt NzInt -> NzNat [comm] .
  vars I J : Int .   vars I' J' : NzInt .
  vars N' M' : NzNat .
  eq - - I = I .
  eq - 0 = 0 .
  eq I + 0 = I .
  eq M' + (- N') = if N' == M' then 0 else
     (if N' > M' then - d(N',M') else d(N',M') fi) fi .
  eq (- I) + (- J) = - (I + J) .
  eq I * 0 = 0 .
  eq 0 * I = 0 .
  eq I * (- J) = - (I * J) .
  eq (- J) * I = - (I * J) .
  eq quot(0,I') = 0 .
  eq quot(- I',J') = - quot(I',J') .
  eq quot(I',- J') = - quot(I',J') .
```

```
  eq gcd(- I',J') = gcd(I',J') .
endo

obj RAT is
  sorts Rat NzRat .
  protecting INT .
  subsort Int < Rat .
  subsorts NzInt < NzRat < Rat .
  op _/_ : Rat NzRat -> Rat .
  op _/_ : NzRat NzRat -> NzRat .
  op -_  : Rat -> Rat .
  op -_  : NzRat -> NzRat .
  op _+_ : Rat Rat -> Rat [assoc comm] .
  op _*_ : Rat Rat -> Rat .
  op _*_ : NzRat NzRat -> NzRat .
  vars I' J' : NzInt .   vars R S : Rat .
  vars R' S' : NzRat .
  eq R / (R' / S') = (R * S') / R' .
  eq (R / R') / S' = R / (R' * S') .
  cq J' / I' = quot(J',gcd(J',I')) / quot(I',gcd(J',I'))
     if gcd(J',I') =/= s 0 .
  eq R / s 0 = R .
  eq 0 / R' = 0 .
  eq R / (- R') = (- R) / R' .
  eq - (R / R') = (- R) / R' .
  eq R + (S / R') = ((R * R') + S) / R' .
  eq R * (S / R') = (R * S) / R' .
  eq (S / R') * R = (R * S) / R' .
endo

obj CPX-RAT is
  sorts Cpx Imag NzImag NzCpx .
  protecting RAT .
  subsort Rat < Cpx .
  subsort NzRat < NzCpx .
  subsorts NzImag < NzCpx Imag < Cpx .
  subsorts Zero < Imag .
  op _i : Rat -> Imag .
  op _i : NzRat -> NzImag .
  op -_ : Cpx -> Cpx .
  op -_ : NzCpx -> NzCpx .
  op _+_ : Cpx Cpx -> Cpx [assoc comm] .
  op _+_ : NzRat NzImag -> NzCpx [assoc comm] .
  op _*_ : Cpx Cpx -> Cpx .
  op _*_ : NzCpx NzCpx -> NzCpx .
  op _/_ : Cpx NzCpx -> Cpx .
  op _# : Cpx -> Cpx .
  op |_|^2 : Cpx -> Rat .
  op |_|^2 : NzCpx -> NzRat .
  vars R S : Rat .   vars R' R'' S' S'' : NzRat .
  vars A B C : Cpx .
  eq 0 i = 0 .
  eq C + 0 = C .
  eq (R i) + (S i) = (R + S) i .
  eq -(R' + (S' i)) = (- R') + ((- S') i ) .
  eq -(S' i) = (- S') i .
  eq R * (S i) = (R * S) i .
  eq (S i) * R = (R * S) i .
```

```
  eq (R i) * (S i) = - (R * S) .
  eq C * (A + B) = (C * A) + (C * B) .
  eq (A + B) * C = (C * A) + (C * B) .
  eq R # = R .
  eq (R' + (S' i))# = R' + ((- S') i) .
  eq (S' i) # = ((- S') i) .
  eq | C |^2 = C * (C #) .
  eq (S' i) / R'' = (S' / R'') i .
  eq (R' + (S' i)) / R'' = (R' / R'') + ((S' / R'') i) .
  eq A / (R' i) = A * (((- s 0) /  R') i) .
  eq A / (R'' + (R' i)) =
     A *((R'' / |(R'' + (R' i))|^2) + (((- R') / |(R'' + (R' i))|^2) i)).
endo

obj QUAT-RAT is
  sorts Quat NzQuat J NzJ .
  protecting CPX-RAT .
  subsorts NzJ Zero < J < Quat .
  subsorts NzCpx < NzQuat Cpx < Quat .
  subsort NzJ < NzQuat .
  op _j : Cpx -> J .
  op _j : NzCpx -> NzJ .
  op -_ : Quat -> Quat .
  op _+_ : Quat Quat -> Quat [assoc comm] .
  op _+_ : Cpx NzJ -> NzQuat [assoc comm] .
  op _*_ : Quat Quat -> Quat .
  op _*_ : NzQuat NzQuat -> NzQuat .
  op _/_ : Quat NzQuat -> Quat .
  op _# : Quat -> Quat .
  op |_|^2 : Quat -> Rat .
  op |_|^2 : NzQuat -> NzRat .
  vars O P Q : Quat .   vars B C : Cpx .
  vars C' : NzCpx .
  eq 0 j = 0 .
  eq Q + 0 = Q .
  eq -(C + (B j)) = (- C) + ((- B) j ) .
  eq (C j) + (B j) = (C + B) j .
  eq C * (B j) = (C * B) j .
  eq (B j) * C = (B * (C #)) j .
  eq (C j) * (B j) = - (C * (B #)) .
  eq Q * (O + P) = (Q * O) + (Q * P) .
  eq (O + P) * Q = (O * Q) + (P * Q) .
  eq (P + Q) # = (P #) + (Q #) .
  eq (C j) # = (- C) j .
  eq | Q |^2 = Q * (Q #) .
  eq Q / (C' j) = Q * ((s 0 / (- C')) j) .
  eq Q / (C + (C' j)) = Q * (((C #) / |(C + (C' j))|^2) +
     (((- C') / |(C + (C' j))|^2) j)) .
endo

obj TST is
  protecting QUAT-RAT .
  ops 1 2 3 4 5 6 7 8 9 : -> NzNat [memo] .
  eq 1 = s 0 .  eq 2 = s 1 .  eq 3 = s 2 .
  eq 4 = s 3 .  eq 5 = s 4 .  eq 6 = s 5 .
  eq 7 = s 6 .  eq 8 = s 7 .  eq 9 = s 8 .
endo
reduce 3 + 2 .
```

```
reduce 3 * 2 .
reduce p p 3 .
reduce 4 > 8 .
reduce d(2,8) .
reduce quot(7,2) .
reduce gcd(9,6) .
reduce (- 4) + 8 .
reduce (- 4) * 2 .
reduce 8 / (- 2) .
reduce (1 / 3) + (4 / 6) .
reduce | 1 + (2 i) |^2 .
reduce | (1 + (3 i)) + (1 + ((- 2) i)) |^2 .
reduce (3 + ((3 i) + ((- 2) i))) / ((2 i) + 2) .
reduce (2 + ((3 i) j)) * ((5 i) + (7 j)) .
reduce (1 + ((1 i) j)) / (2 j) .
```

C.8 Categories and Coproducts

This subsection specifies categories and coproducts. Some familiarity with category theory may be needed (e.g., sections 2.3 and 3.9 of [88]); on the other hand, the code may also provide a more concrete understanding of the categorical concepts; see also [136]. Note how universal morphisms are defined as a subsort, and also the use of sort constraints. (Recall that the semantics of `op-as` is not yet implemented.) The use of `memo` has quite a significant effect on performance in this example.

```
*** theory of categories
th CAT-TH is
  sorts Mor Obj .
  ops (d0_) (d1_) : Mor -> Obj .
  op-as _;_ : Mor Mor -> Mor for M1 ; M2 if d1 M1 == d0 M2 [assoc] .
  op id_ : Obj -> Mor .
  var O : Obj .
  vars M M0 M1 : Mor .
  eq d0 id O = O .
  eq d1 id O = O .
  eq d0 (M0 ; M1) = d0 M0 .
  eq d1 (M0 ; M1) = d1 M1 .
  eq (id d0 M); M = M .
  eq M ; id d1 M = M .
endth

*** generic category of sets
obj CAT-SET[X :: TRIV] is
  sort Fn .
  protecting SET[X] .
  ops (d0_) (d1_) : Fn -> Set .
  op-as _;_ : Fn Fn -> Fn for F1 ; F2 if d1 F1 == d0 F2 [assoc] .
  op id_ : Set -> Fn .
  op-as _of_ : Fn Elt -> Elt for F of X if (X in d0 F) and (F of X in d1 F) .
  var O : Set .
  vars F F0 F1 : Fn .
  var E : Elt .
  eq d0 id O = O .
  eq d1 id O = O .
  eq d0 (F0 ; F1) = d0 F0 .
  eq d1 (F0 ; F1) = d1 F1 .
  eq (id d0 F) ; F = F .
  eq F ; id d1 F = F .
  eq (F0 ; F1) of E = F0 of (F1 of E) .
  eq (id O) of E = E .
```

```
endo

*** CAT-SET always gives a category
view CAT-SET-AS-CAT from CAT-TH to CAT-SET is
  sort Obj to Set .
  sort Mor to Fn .
endv

*** 2-cones in C
obj CO2CONE[C :: CAT-TH] is
  sort Co2cone .
  define Base is 2TUPLE[Obj,Obj] .
  op-as cone : Mor Mor -> Co2cone for cone(M1,M2) if d1 M1 == d1 M2 .
  ops j1 j2 : Co2cone -> Mor .
  op apex : Co2cone -> Obj .
  op base : Co2cone -> Base .
  vars M1 M2 : Mor .
  eq j1(cone(M1,M2)) = M1 .
  eq j2(cone(M1,M2)) = M2 .
  eq apex(cone(M1,M2)) = d1 M1 .
  eq base(cone(M1,M2)) = << d0 M1 ; d0 M2 >> .
endo

*** theory of coproduct in C
th CO2PROD-TH[C :: CAT-TH] is
  sort Uco2cone .
  protecting CO2CONE[C] .
  subsort Uco2cone < Co2cone .      *** a very nice subsort!
  op ucone : Obj Obj -> Uco2cone .  *** coproduct cone
  op _++_ : Obj Obj -> Obj .        *** coproduct object
  op-as umor : Uco2cone Co2cone -> Mor for umor(U,C) if base(U) == base(C) .
  vars A B : Obj .
  vars F G H : Mor .
  eq apex(ucone(A,B)) = A ++ B .
  eq base(ucone(A,B)) = << A ; B >> .
  eq (j1(ucone(A,B))); umor(ucone(A,B),cone(F,G)) = F .
  eq (j2(ucone(A,B))); umor(ucone(A,B),cone(F,G)) = G .
  cq H = umor(ucone(A,B),cone(F,G))
     if (j1(ucone(A,B)); H == F) and (j2(ucone(A,B)); H == G) .
endth

*** theory of injections for building a coproduct
th 2INJ-TH is
  sort Elt .
  ops i0 i1 i0inv i1inv : Elt -> Elt .
  ops i0pred i1pred : Elt -> Bool .
  var E : Elt .
  eq i0inv(i0(E)) = E .
  eq i1inv(i1(E)) = E .
  eq i0pred(i0(E)) = true .
  eq i0pred(i1(E)) = false .
  eq i1pred(i1(E)) = true .
  eq i1pred(i0(E)) = false .
endth

*** coproduct in a category of sets, given injections for it
obj CO2PROD-CAT-SET[J :: 2INJ-TH] is
  sort Uco2cone .
```

```
  extending CO2CONE[view to CAT-SET[J] is
      sort Obj to Set .
      sort Mor to Fn .
    endv] .
  subsort Uco2cone < Co2cone .
  op ucone : Set Set -> Uco2cone .
  op-as umor : Uco2cone Co2cone -> Fn for umor(U,C) if base(U) == base(C) .
  ops I0 I1 : Set -> Set .
  op _++_ : Set Set -> Set [memo] .
  vars A B S : Set .
  vars F G : Fn .
  var E : Elt .
  eq I0({}) = {} .
  eq I0({ E } + S) = { i0(E) } + I0(S) .
  eq I1({}) = {} .
  eq I1({ E } + S) = { i1(E) } + I1(S) .
  eq A ++ B = I0(A) U I1(B) .
  eq apex(ucone(A,B)) = A ++ B .
  eq base(ucone(A,B)) = << A ; B >> .
  cq j1(ucone(A,B)) of E = i0(E) if E in A .
  cq j2(ucone(A,B)) of E = i1(E) if E in B .
  cq umor(ucone(A,B),cone(F,G)) of E = F of i0inv(E) if i0pred(E) .
  cq umor(ucone(A,B),cone(F,G)) of E = G of i1inv(E) if i1pred(E) .
endo

*** CO2PROD-CAT-SET gives a coproduct
*** view CO2PROD-CAT-SET-AS-CO2PROD-TH[J :: 2INJ-TH]
***    from CO2PROD-TH[CAT-SET[J]] to CO2PROD-CAT-SET[J] endv
*** don't have parameterized views yet

*** constructions for the category of sets of integers
make CAT-SET-INT is CAT-SET[INT]*(op omega to ints) endm

*** coproduct in the category of sets of integers
make CO2PROD-CAT-SET-INT is
  CO2PROD-CAT-SET[view to INT is
      var I : Elt .
      op i0(I) to (2 * I) .
      op i0inv(I) to (I quo 2) .
      op i0pred(I) to (I rem 2 == 0) .
      op i1(I) to 1 + (2 * I) .
      op i1inv(I) to ((I - 1) quo 2) .
      op i1pred(I) to (I rem 2 == 1) .
    endv]
endm

*** this says the above really is a coproduct
view CO2PROD-CAT-SET-INT-VIEW from CO2PROD-TH[view to CAT-SET[INT] is
       sort Obj to Set .
       sort Mor to Fn .  endv]
     to CO2PROD-CAT-SET-INT is endv
*** note the view within view and empty body of outermost view

*** some test cases
obj CO2PROD-TEST is
  protecting CO2PROD-CAT-SET-INT .
  ops s1 s2 s3 s4 : -> Set [memo] .
  eq s1 = { 1 } .
```

```
  eq s2 = s1 U { 2 } .
  eq s3 = s2 U { 3 } .
  eq s4 = { 2 } U { 3 } .
  op g : -> Fn .
  eq g of 3 = 2 .
  eq g of 2 = 1 .
  eq d0 g = s4 .
  eq d1 g = s3 .
endo

reduce base(ucone(s1,s1)) .                          ***> should be: <<{1};{1}>>
reduce apex(ucone(s1,s1)) .                          ***> should be: {2,3}
reduce umor(ucone(s1,s1),cone(id s1,id s1)) of 2 .  ***> should be: 1
reduce umor(ucone(s1,s1),cone(id s1,id s1)) of 3 .  ***> should be: 1

reduce base(ucone(s2,s3)) .                       ***> should be: <<{1,2};{1,2,3}>>
reduce apex(ucone(s2,s3)) .                       ***> should be: {2,4,3,5,7}
reduce umor(ucone(s2,s3),cone(id s2,id s3)) of 2 .  ***> should be: 1
reduce umor(ucone(s2,s3),cone(id s2,id s3)) of 4 .  ***> should be: 2
reduce umor(ucone(s2,s3),cone(id s2,id s3)) of 3 .  ***> should be: 1
reduce umor(ucone(s2,s3),cone(id s2,id s3)) of 5 .  ***> should be: 2
reduce umor(ucone(s2,s3),cone(id s2,id s3)) of 7 .  ***> should be: 3

reduce base(ucone(s4 ,s3)) .                      ***> should be: <<{2,3};{1,2,3}>>
reduce apex(ucone(s4,s3)) .                       ***> should be: {4,6,3,5,7}
reduce umor(ucone(s4,s3),cone(g,id s3)) of 4 .    ***> should be: 1
reduce umor(ucone(s4,s3),cone(g,id s3)) of 6 .    ***> should be: 2
reduce umor(ucone(s4,s3),cone(g,id s3)) of 3 .    ***> should be: 1
reduce umor(ucone(s4,s3),cone(g,id s3)) of 5 .    ***> should be: 2
reduce umor(ucone(s4,s3),cone(g,id s3)) of 7 .    ***> should be: 3
```

D Built-ins and the Standard Prelude

This Appendix gives details of how Lisp can be called from within OBJ3 programs, with a number of examples, including the complete OBJ3 standard prelude.

D.1 The Lisp Interface

OBJ3 provides two ways to take advantage of the (Common-)Lisp underlying its implementation: *built-in sorts* and *built-in righthand sides* for rules; we call rules with such built-in righthand sides *built-in rules*.

Built-in sorts are sorts whose elements are constants represented by Lisp values. General mechanisms are provided for reading, printing, creating Lisp representations for, and testing sort membership for constants of these sorts. In general, built-in sorts can be used in any context where a non-built-in sort can be used, although a constant of a built-in sort cannot be the lefthand side of a rule.

The built-in rules come in two varieties, a simplified version that makes writing rules for operators defined on built-in sorts easy, and a general kind that allows arbitrary actions on the redex to be specified. However, to take full advantage of this latter type of rule, one must be familiar with the internal term representation of OBJ3 and the implementation functions for manipulating this representation. Built-in rules can be used wherever an ordinary rule can be.

D.1.1 Built-in Sorts

Built-in sorts may contain any (countable) number of constants. For example, a version of NATS with a built-in sort Nat is equivalent to an idealized non-built-in version of the form

```
obj NATS is
  sort Nat .
  ops 0 1 2 3 4 5 6 7 8 9 10 11 12 13 ... : -> Nat .
  op _+_ : Nat Nat -> Nat .
```

```
  *** etc.
endo
```

with an infinite number of constants. (The name `NATS` is chosen to avoid clashing with the predefined object `NAT`.) Some other useful built-in sorts are floating-point numbers, identifiers, strings, and arrays.

Constants in a built-in sort have an associated Lisp representation. Such a built-in sort is introduced by a declaration of the form

`bsort` ⟨*SortId*⟩ (⟨*Token-Predicate*⟩ ⟨*Creator*⟩ ⟨*Printer*⟩ ⟨*Sort-Predicate*⟩) .

which gives the name of the sort, two Lisp functions for reading, a function for printing constants of the sort, and a predicate that can be used to test whether a Lisp value represents a constant of the given sort. A sort declaration of this kind can occur wherever an ordinary sort declaration can occur.

When an OBJ expression is read, it is first lexically analyzed into a sequence of tokens that are either special single character symbols, such as "`(`" and "`]`", or else are sequences of characters delimited by these special single character symbols or spaces. Internally, such tokens are represented by Lisp strings. For example, the representation of the token "`37`" is the Lisp string `"37"` of length two.

In more detail now:

- ⟨*Token-Predicate*⟩ is a Lisp predicate that can be applied to an input token (a Lisp string) to determine if the token is a representation of a value in the built-in sort (it is applied by `funcall`); for example, `"37"` from `NATS` should result in `true` and `"A+B"` in `false`. With this mechanism, the syntactic representation of a built-in constant can only be a single token.
- ⟨*Creator*⟩ is a Lisp function that will map a token (a Lisp string) to a Lisp representation for that token as a built-in constant. The Lisp function `read-from-string` is very useful as a creator function for built-in sorts that correspond directly to Lisp types. For example, `"37"` should be mapped to the Lisp value `37`.
- ⟨*Printer*⟩ is a Lisp function that will print out the desired external representation of the internal Lisp value representing one of the built-in sort constants. The Lisp function `prin1` is very useful as a ⟨*Printer*⟩ function for printing out values that correspond directly to Lisp types. For example, `37` should be printed by printing the digit `3` followed by the digit `7`. Because the user can define the printer function to meet particular needs, there is no assumption that this function is an inverse to the ⟨*Creator*⟩ function. Indeed, the syntactic representation of a built-in constant may involve many tokens, but then this representation cannot be read in as a built-in constant.
- ⟨*Sort-Predicate*⟩ is a Lisp predicate that is true only for Lisp values that are representations of constants in the built-in sort. For example, `3` should be considered to be in sort `Nat`, and `-3` should not. The purpose and use of this predicate are discussed further below.

For example, to define `NATS` we might first define some Lisp functions, with

```
ev (progn
(defun obj_NATS$is_Nat_token (token)
  (every #'digit-char-p token))
(defun obj_NATS$create_Nat (token) (read-from-string token))
(defun obj_NATS$print_Nat (x) (prin1 x))
(defun obj_NATS$is_Nat (x) (and (integerp x) (<= 0 x))))
```

Then we can define

```
obj NATS is
  bsort Nat (obj_NATS$is_Nat_token obj_NATS$create_Nat
             obj_NATS$print_Nat obj_NATS$is_Nat) .
endo
```

which supports the natural number constants, as in

```
OBJ> red 100 .
reduce in NATS : 100
rewrites: 0
result Nat: 100
```

However, this object is not very useful, because no operators have been associated to the built-in sort.

Warning: A current implementation restriction does not allow a built-in constant to be the lefthand side of a rule. Built-in constants are always considered to be in reduced form, so that rule applications are never attempted on them.

D.1.2 Subsorts of Built-in Sorts

It is possible for a built-in sort to be a subsort of another built-in sort, but a non-built-in sort cannot be a subsort of a built-in sort. However, a non-built-in sort can be a supersort of a built-in sort. For the sort of newly created built-in constants to be properly assigned, a sort predicate must be provided for each built-in sort. An example of this will later be seen in a version of the rational numbers using Common Lisp rationals.

When there are built-in subsorts of the sort of a newly created built-in constant, then the sort that is assigned to the constant is determined by scanning the list of subsorts, applying the sort predicates to the Lisp value to determine if it lies in the corresponding subsort, and choosing the lowest acceptable sort as the sort of the constant. It is assumed that there is always a unique lowest sort. It is critical only that the sort predicate for a built-in sort should be false for values that are in supersorts of the built-in sort. It is not necessary for it to be false for constants in subsorts of the given sort.

If there is no enclosing built-in supersort, then it can be the constant true, and have a definition like

```
(defun obj_NATS$is_Nat (x) t)
```

because the ⟨*Sort-Predicate*⟩ function will only be called for built-in constants of that sort (if it is called at all). This will not affect the operational behavior of OBJ in this case. However, it is better for the predicate to be exact in order to allow the easy incorporation of new supersorts.

D.1.3 Built-in Rules

Built-in rules provide a way of using Lisp expressions to perform computations. These are essential for the usefulness of built-in sorts, but they can also be used for non-built-in data. Built-in rules are either of a special *simple* form or else are *general*.

Simple built-in rules can be unconditional or conditional, with the syntax

```
bq ⟨Term⟩ = ⟨Lisp Expression⟩ . |
cbq ⟨Term⟩ = ⟨Lisp Expression⟩ if ⟨BoolTerm⟩ .
```

The key restriction on simple built-in rules is that *the sort of each variable appearing in the lefthand side must be a built-in sort*.

The lefthand side of these rule is matched against terms in exactly the usual fashion; also, in the conditional case, the condition is just an OBJ term, and it is treated in exactly the same way as a condition in a non-built-in rule. If a match is found for the variables in the lefthand side such that each variable matches a built-in constant (and the condition is satisfied if the built-in rule is conditional), then the righthand side is evaluated in a Lisp environment with Lisp variables having names corresponding to the OBJ variables (as usual in Common Lisp, the case, upper or lower, of variables in the Lisp expression is ignored) bound to the Lisp value of the built-in constants to which they were matched. Because the variables must match constants of the corresponding built-in sorts, a bottom-up evaluation strategy is necessary, regardless of the strategy specified for the operator. The sort of the lefthand side is usually a built-in sort, and in this case the Lisp value of the righthand side of the rule is automatically converted to a built-in constant of that sort. If the sort of the lefthand side is not a built-in sort, then, with one exception that will be mentioned next, the value of the righthand side should be a Lisp representation of a term of that sort (or a subsort of that sort). A special case is that, if the sort of the lefthand side is `Bool`, then the value of the righthand side Lisp expression can be any Lisp value that will be converted to a Boolean value by mapping `nil` to `false` and all other Lisp values to `true`. For this case, a special conversion is performed; this makes it very easy to define predicates.

As an example, consider

```
obj NATS is
  bsort Nat (obj_NATS$is_Nat_token obj_NATS$create_Nat
             obj_NATS$print_Nat obj_NATS$is_Nat) .
  op _+_ : Nat Nat -> Nat .
```

```
  vars M N : Nat .
  bq M + N = (+ M N) .
endo
```

We can then do the following reduction.

```
OBJ> red 123 + 321 .
reduce in NATS : 123 + 321
rewrites: 1
result Nat: 444
```

Because the matching of the lefthand side is done in the usual fashion, the operators appearing in the lefthand side can even be associative and commutative.

The *general* form of a built-in rule has the following syntax

bq ⟨*Term*⟩ = ⟨*Lisp Expression*⟩ . |
cbq ⟨*Term*⟩ = ⟨*Lisp Expression*⟩ if ⟨*BoolTerm*⟩ .

where now the variables in the lefthand side can have arbitrary sorts. The lefthand side and condition are treated as usual.

The process of applying the rule is a bit different in this case. The lefthand side is matched as usual creating the correspondence between variables in the lefthand side and subterms of the term being rewritten. The righthand side is evaluated in an environment where Lisp variables with names corresponding to the OBJ variables (case is ignored) are bound to the internal OBJ3 representation of the terms matched by the variables. The Lisp value of the righthand side is expected to be an internal OBJ3 representation of a term that then destructively replaces the top-level structure of the term matched. An exception is that, if the Lisp code evaluates the expression `(obj$rewrite_fail)`, then the rewrite is aborted and the term is left unchanged. (This has the effect of making the rule conditional in an implicit way; the condition is checked in the Lisp code for the righthand side.) An additional feature is that the righthand side is evaluated in an environment where `module` is bound to the module that the rule comes from; this feature is necessary to correctly treat general built-in rules in instances of parameterized modules.

The following is a simple example:

```
obj NATS is
  bsort Nat (obj_NATS$is_Nat_token obj_NATS$create_Nat
             obj_NATS$print_Nat obj_NATS$is_Nat) .
  op _+_ : Nat Nat -> Nat .
  vars M N : Nat .
  bq M + N = (+ M N) .
  op print _ : Nat -> Nat .
  beq print M = (progn (princ " = ") (term$print M) (terpri) M) .
endo
```

The operator `print` returns just its argument, but has the side-effect of printing the term resulting from evaluating its argument preceded by the "=" sign. For example,

```
red (print (3 + 2)) + 4 .
```

produces the following output from OBJ3:

```
==========================================
reduce in NATS : print (3 + 2) + 4
 = 5
rewrites: 3
result Nat: 9
==========================================
```

The line containing "= 5" is the output produced by the use of `print`. Such print operators can be very useful; in many cases, one may want to add an extra argument that provides an output label. General built-in rules can be written to perform arbitrary transformations on a term using any of the functions defined in the OBJ3 implementation. Thus it is useful to be familiar with the functions provided by the implementation when writing such general built-in rules. Some basic functions are discussed below.

It is often useful to initialize some Lisp variables after certain OBJ objects are created. This can be done using `eval` or `ev`. There are examples of this in the OBJ3 standard prelude.

In general, the module that the rules are associated with may be an instance of a parameterized module. In this case, it is necessary to write the rules so that the extra parameter `module` is used to create structures within that module. When locating the correct instance of an operator one must first determine its module, then the sorts of its arguments and result, and then its name. In the case where there are no ambiguities, some simpler functions can be used, e.g., to find an operator based only on its name.

Functions that are useful for the general built-in rules are given below (note that these are all Lisp functions from the OBJ3 implementation). The Lisp functions will be described, in part, by giving declarations similar to OBJ operator declarations. Of course these need to be interpreted as informal descriptions of Lisp functions that may have side-effects and that manipulate particular Lisp representations of the values given as arguments.

The sorts that will be referred to are:

- `Bool`, `NzNat`, `Lisp-Value`
  `Bool`, `NzNat`, and `Lisp-Value` are OBJ sort names for the related standard Lisp types.

- `Sort-Name`
  A `Sort-Name` is a Lisp string naming a sort.

- `Op-Name`
  An `Op-Name` is a Lisp list of the tokens, represented as Lisp strings, that constitute the name of the operator. For example, the name of `_+_ : Nat Nat -> Nat` is `("_" "+" "_")`.

- Sort-Order
  A `Sort-Order` is a representation of a partial order on the sorts.

- `Sort`, `Operator`, `Term`, `Module`, `Module-Expression`
  `Sort`, `Operator`, `Term`, `Module`, and `Module-Expression` correspond to the Lisp representations of these sorts. Values of the sorts `Sort`, `Operator`, `Term`, and `Module` are composite objects with many components, some of which are likely not to be of interest here. For these sorts, functions selecting the interesting features of the values are given below.

- `SortSet`
  `SortSet` is a set of sorts represented by a list.

- `LIST[Term]`, `LIST[Sort]`
  `LIST[–]` indicates that the values so described will be Lisp lists of the specified sort.

The following functions are useful for term manipulation:

- `modexp_eval$eval` : `Module-Expression` → `Module`
  The argument can be the name of a specific named module, such as `"INT"`. This can be used to find specific named modules.

- `sort$is_built_in` : `Sort` → `Bool`
  This predicate decides whether the sort given is a built-in sort.

- `module$sort_order` : `Module` → `Sort-Order`
  This selector provides access to the sort order for the given module, i.e., the representation of the sort structure.

- `sort_order$is_included_in` : `Sort-Order Sort Sort` → `Bool`
  This predicate decides if the first sort is a subsort of the second in the given sort order.

- `sort_order$is_strictly_included_in` : `Sort-Order Sort Sort` → `Bool`
  Same as above but excludes the case when two sorts are equal.

- `sort_order$lower_sorts` : `Sort-Order Sort` → `SortSet`
  This function produces a list of the sorts lower than a given sort in the given sort order.

- mod_eval$$find_sort_in : Module Sort-Name → Sort
  This function can be used to find the named sort in the given module. A typical sort name is “"Int"”.
- sort$name : Sort → Sort-Name
  This selector provides the name of a given sort.
- operator$name : Operator → Op-Name
  This selector provides the name of the given operator.
- operator$is_same_operator : Operator Operator → Bool
  This predicate decides if the two operators are the same operator.
- operator$arity : Operator → LIST[Sort]
  This selector provides the arity of the given operator as a list of sorts, which may be nil.
- operator$coarity : Operator → Sort
  This selector provides the co-arity of the given operator.
- mod_eval$$find_operator_in : Module Op-Name LIST[Sort] Sort → Operator
  This function locates the operator with the given name, arity (list of sorts) and coarity (value sort), or returns nil if there is none such.
- mod_eval$$find_operator_named_in : Module Op-Name → Operator
  This function attempts to locate an operator purely based on its name.
- term$is_var : Term → Bool
  This predicate decides if a term is a variable. It may be that the terms that you are manipulating are primarily be ground terms, but, in general, it is preferable to consider the case of variables in the definitions of functions.
- term$is_constant : Term → Bool
  This predicate decides if a term is a constant.
- term$head : Term → Operator
  This function produces the operator that is the head operator of a non-variable term. It is an error to apply this function to a term that is a variable.
- term$subterms : Term → LIST[Term]
  This function produces the list of top-level subterms of the given term.
- term$make_term : Operator LIST[Term] → Term
  This function creates a new term with the given head operator and list of arguments.
- term$make_term_with_sort_check : Operator LIST[Term] → Term
  This function is similar to the last, but may replace the operator with a lower operator in the case of overloading. If there is a lower overloaded operator whose arity fits the sorts of the given arguments, then this operator will be used instead of the given operator.
- term$arg_n : Term NzNat → Term
  This function gives easy access to the n-th (counting from 1) top-level argument of the given term.
- term$sort : Term → Sort
  This function computes the sort of a term whether or not it is a variable.
- term$is_reduced : Term → Bool
  This function checks whether or not the term has been marked as fully reduced. This flag is updated by side-effect.
- term$!replace : Term Term → Term
  The Lisp representation for the first argument term is destructively altered in such a way that it will appear to have the same term structure as the second term argument. The altered representation of the first term is returned.
- term$!update_lowest_parse_on_top : Term → Term
  This will update the sort of the term, for example, when a subterm has been altered so that it now has a lower sort.

- `term$retract_if_needed` : `Sort-Order Term Sort` → `Term`
  This function either returns the term, or a retract applied to the term depending on whether or not the sort of the term is included in the given sort.

- `term$is_built_in_constant` : `Term` → `Bool`
  This predicate decides if the term is a built-in constant or not.

- `term$similar` : `Term Term` → `Bool`
  Tests if the two terms have the same term structure without taking attributes into account.

- `term$equational_equal` : `Term` → `Bool`
  Tests if the terms have the equivalent structure taking attributes into account.

- `term$make_built_in_constant` : `Sort Lisp-Value` → `Term`
  This function creates a *term* which is a built-in constant for the given built-in sort and Lisp value. The sort predicate for the built-in sort is not applied.

- `term$make_built_in_constant_with_sort_check` : `Sort Lisp-Value` → `Term`
  Similar to above, but may replace the given sort by a lower sort.

- `term$built_in_value` : `Term` → `Lisp-Value`
  This function produces the Lisp value from a built-in constant.

- `obj_BOOL$is_true` : `Term` → `Bool`
  This function tests whether the term given as its argument is the constant `true`. The value is a Lisp boolean, i.e. `T` for `true` and `NIL` for `false`.

- `rew$!normalize` : `Term` → `Term`
  This is the OBJ evaluation function. The term given as an argument is reduced and is updated by side-effect as well as being returned as the value of the function.

The following functions are specific to terms where the top operator is associative (A) or associative-commutative (AC):

- `term$list_assoc_subterms` : `Term Operator` → `LIST[Term]`
  This function computes the list of subterms of the given term that are on the fringe of the tree at the top of the term the nodes of which are all terms headed with the given associative operator or operators overloaded by this operator. This can be the whole term.

- `term$list_AC_subterms` : `Term Operator` → `LIST[Term]`
  Similar to the above, but for associative-commutative (AC) operators.

- `term$make_right_assoc_normal_form` : `Operator LIST[Term]` → `Term`
  This function builds a term from the given associative operator and the list of terms by building a right-associated binary tree.

- `term$make_right_assoc_normal_form_with_sort_check` : `Operator LIST[Term]` → `Term`
  Similar to the above, but may replace the operator by lower operators.

The predefined object `BUILT-IN` (see Section D.3) allows the creation of built-in subterms of righthand sides of rules. The default syntax is "`built-in:` ⟨*Lisp*⟩", where the Lisp expression represents a function, to be `funcall`-ed, that takes one argument, which is a substitution, and produces two values, a term which is the intended instantiation for this subterm, and a success indicator . In general, it will be necessary to deal with the incompatibility of the sort `Built-in` with the sorts of other operators in the righthand side. Here is a sketch of a use of this feature:

```
op r : Universal -> A .
var X : A .
eq f(X) = X + r(built-in: (lambda (u) (create-term u))) .
eq r(X) = X .
```

Note that "`built-in:`" is now a very special keyword, and cannot be used in any other context (this can be disabled by "`ev (setq obj_BUILT-IN$keyword nil)`").

D.2 Examples

We now give a number of somewhat larger examples, including cells (which have internal memory), arrays of integers, and an efficient sorting program. Other examples appear in the standard prelude, although the Lisp code used in the standard prelude is not given in this paper.

D.2.1 Cells

The basic idea of this example is very simple, namely to provide a parameterized object that creates *cells* containing values of a given sort. Such cells are an abstract version of procedural variables that can be modified by side-effects or destructive assignments. Of course, this module is not functional.

```
*** obj code for cells

ev (defun set-cell-rule (i x) (setf (cadr i) x) i)

obj CELL[X :: TRIV] is
  sort Cell .
  op cell _ : Elt -> Cell .
  op new-cell _ : Elt -> Cell .
  op val _ : Cell -> Elt .
  op set _ _ : Cell Elt -> Cell .
  var I : Cell .
  var X : Elt .
  eq new-cell X = cell X .
  eq val (cell  X) = X .
  beq set I X = (set-cell-rule I X) .
endo

*** sample program using this
obj TEST is
  pr CELL[INT] .

  sort A .
  subsort Int Cell < A .
  op _|_ : A A -> A .

  op dbl _ : A -> A .
  op incr _ : A -> A .

  var U V : A .
  var C : Cell .

  eq dbl U = U | U .

  eq incr (U | V) = (incr U) | (incr V) .
  eq incr C = val (set C (1 + (val C))) .
endo

red incr (dbl (dbl (dbl (new-cell 0)))) .
*** result A: ((1 | 2) | (3 | 4)) | ((5 | 6) | (7 | 8))
```

D.2.2 Arrays of Integers

The following code provides arrays of integers that can be modified by side-effect. This might be useful for a functional program for table-lookup (side-effects would only be used for building the table).

```
ev
(defun arrayint$print (x)
  (princ "[")
```

```
  (dotimes (i (length x))
    (when (< 0 i) (princ ",")) (print$check)
    (prin1 (aref x i)))
  (princ "]"))

obj ARRAYINT is
  pr INT .
  bsort ArrayInt ((lambda (x) nil) (lambda (x) (break))
                  arrayint$print (lambda (x) t)) .

  op make-array : Nat Int -> ArrayInt .
  op length _ : ArrayInt -> Nat .
  op _[_] : ArrayInt Nat -> Int .
  op _[_] := _ : ArrayInt Nat Int -> ArrayInt .

  var A : ArrayInt .
  var I : Int .
  var N : Nat .

  bq make-array(N,I) = (make-array (list N) :initial-element I) .
  bq length(A) = (length A) .
  bq A[N] = (aref A N) .
  bq A[N] := I = (progn (setf (aref A N) I) A) .
endo
```

The commands

```
red make-array(10,1) .
red (make-array(10,1))[5] .
red (make-array(10,1))[5] := 33 .
```

produce the following output:

```
==========================================
reduce in ARRAYINT : make-array(10,1)
rewrites: 1
result ArrayInt: [1,1,1,1,1,1,1,1,1,1]
==========================================
reduce in ARRAYINT : make-array(10,1)[5]
rewrites: 2
result NzNat: 1
==========================================
reduce in ARRAYINT : make-array(10,1)[5]:= 33
rewrites: 2
result ArrayInt: [1,1,1,1,1,33,1,1,1,1]
==========================================
```

D.2.3 Sorting

This example defines a parameterized sorting module. The parameter provides the partial order used and the sorting is done using the Lisp function `sort`. A small point of some interest is that an operator named `_<<_` is introduced as an alias for the parameter operator `_ < _` simply to provide an easy way to locate the parameter operator after instantiation. This is needed because the name of a parameter operator cannot be known for an instance of the parameterized module, where such a parameter may have been mapped to an arbitrary term by the view defining the instantiation. For similar reasons, the operator `_<<_` as well as the other operators `_,_` and `empty` appearing in the parameterized `SORT` module below should not be renamed by a module renaming.

As usual, the parameter of our sorting module is the theory of partially ordered sets:

```
th POSET is
```

```
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars E1 E2 E3 : Elt .
  eq E1 < E1 = false .
  cq E1 < E3 = true if E1 < E2 and E2 < E3 .
endth
```

The Lisp function `sort-list` used in the `SORT` module below has two arguments, a module (namely the given instantiation of the parameterized module `SORT`) and a list to be sorted. Its definition is as follows:

```
ev
; NOTE: sort-list will not work if any of the operators found by name,
; i.e. _<<_, empty, and _,_, below are renamed in a module renaming.

(defun sort-list (mod l)
  (let ((test (mod_eval$$find_operator_named_in
               mod '("_" "<<" "_")))
        (empty (mod_eval$$find_operator_named_in
                mod '("empty")))
        (conc (mod_eval$$find_operator_named_in
               mod '("_" "," "_"))))
  (if (eq empty (term$head l))
      l
    (let ((sorted (sort (term$list_assoc_subterms l conc)
                        #'(lambda (x y)
                            (obj_BOOL$is_true
                             (rew$!normalize
                              (term$make_term test
                                  (list x y)))))
                        )))
      (term$make_right_assoc_normal_form_with_sort_check
       conc sorted)
      ))
  ))
```

We are now ready to define a parameterized sorting module with a built-in equation involving the `sort-list` function:

```
obj SORT[ORDER :: POSET] is
  sort List .
  subsort Elt < List .
  op empty : -> List .
  op _,_ : List List -> List [assoc idr: empty] .
  op sort _ : List -> List .
  op _<<_ : Elt Elt -> Bool .
  vars E1 E2 : Elt .
  eq E1 << E2 = E1 < E2 .
  var L : List .
  beq sort L = (sort-list module L) .
endo
```

Here is a sample reduction for sorting lists of integers.

```
obj TEST is pr SORT[INT] . endo

red sort (9, 8, 7, 6, 5, 4, 3, 2, 1, 0) .
***> result List: 0,1,2,3,4,5,6,7,8,9
```

D.3 The Standard Prelude

Before giving the prelude, we comment on some changes since Release 1 of OBJ3:

- The module `THAT` has been renamed to `LAST-TERM`, and the operator `[that]` has been renamed to `[term]`.
- The module `RAT` has been slightly changed so that the built-in constant values are printed like `1/2`, rather than `1 / 2`, and the same syntax (`1/2`) can be used for the input of these constants. (Previously, there was no syntax for the input of these constants.)
- The object `BUILT-IN` has been modified to allow the creation of built-in subterms of right sides of rules, as discussed in Section D.1.
- A object `LISP` has been added. It provides a built-in Lisp sort. The default syntax is "`lisp:` ⟨*Lisp*⟩". This can be used to allow the use of string data with Lisp syntax for the strings. The keyword that introduces the data (above "`lisp:`") can be changed to be some other symbol by `setq`-ing the variable `obj_LISP$keyword` to that other token (e.g. "`string:`"). Note that "`lisp:`" is now a very special keyword, and cannot be used in any other context (this can be disabled by "`ev (setq obj_LISP$keyword nil)`").

What follows is the exact text of the standard prelude that is used to build OBJ3; it uses many Lisp functions that are not defined here, but rather in another file.

```
--- OBJ standard prelude

ev (setq *obj$include_BOOL* nil)

obj UNIVERSAL is
  sort Universal .
endo

ev (progn (obj_UNIVERSAL$install) 'done)

obj ERR is
 bsort Err
   (obj_ERR$is_Err_token
    obj_ERR$create_Err
    obj_ERR$print_Err
    obj_ERR$is_Err) .
endo

ev (progn (obj_ERR$install) 'done)

obj BUILT-IN is
 bsort Built-in
   (obj_BUILT-IN$is_Built-in_token
    obj_BUILT-IN$create_Built-in
    obj_BUILT-IN$print_Built-in
    obj_BUILT-IN$is_Built-in) .
endo

ev (progn (obj_BUILT-IN$install) 'done)

obj LISP is
 bsort Lisp
   (obj_LISP$is_Lisp_token
    obj_LISP$create_Lisp
    obj_LISP$print_Lisp
    obj_LISP$is_Lisp) .
endo
```

```
obj TRUTH-VALUE is
  sort Bool .
  op false : -> Bool .
  op true : -> Bool .
endo

obj TRUTH is
  pr TRUTH-VALUE .
  pr UNIVERSAL .
  op if_then_else_fi : Bool Universal Universal -> Universal
 [polymorphic obj_BOOL$if_resolver intrinsic strategy (1 0)
  gather (& & &) prec 0] .
  op _==_ : Universal Universal -> Bool
 [polymorphic obj_BOOL$eqeq_resolver strategy (1 2 0) prec 51] .
  op _=/=_ : Universal Universal -> Bool
 [polymorphic obj_BOOL$non-eqeq_resolver strategy (1 2 0) prec 51] .
  ev (obj_TRUTH$setup)
  var XU YU : Universal .
  eq if true then XU else YU fi = XU .
  eq if false then XU else YU fi = YU .
  ev (obj_TRUTH$install)
  beq XU == YU =
      (obj_BOOL$coerce_to_Bool (term$equational_equal XU YU)) .
  beq XU =/= YU =
      (obj_BOOL$coerce_to_Bool (not (term$equational_equal XU YU))) .
  ev (obj_TRUTH$install)
endo

obj BOOL is
  pr TRUTH .
  op _and_ : Bool Bool -> Bool [assoc comm idr: true
strat (1 2 0)
gather (e E) prec 55] .
  op _or_ : Bool Bool -> Bool [assoc comm idr: false
strat (1 2 0)
gather (e E) prec 59] .
  op _xor_ : Bool Bool -> Bool [assoc comm idr: false
strat (1 2 0)
gather (e E) prec 57] .
  op not_ : Bool -> Bool [prec 53] .
  op _implies_ : Bool Bool -> Bool [gather (e E) prec 61] .
  ev (obj_BOOL$setup)
  vars A B : Bool .
  eq not true = false .
  eq not false = true .
  eq false and A = false .
  eq true or A = true .
  eq A implies B = (not A) or B .
  eq true xor true = false .
endo

obj IDENTICAL is
  pr BOOL .
  op _===_ : Universal Universal -> Bool [strategy (0) prec 51] .
  op _=/==_ : Universal Universal -> Bool [strategy (0) prec 51] .
  var XU YU : Universal .
  beq XU === YU =
```

```
      (obj_BOOL$coerce_to_Bool (term$similar XU YU)) .
  beq XU =/== YU =
      (obj_BOOL$coerce_to_Bool (not (term$similar XU YU))) .
endo

ev (progn (obj_IDENTICAL$setup) 'done)

obj LAST-TERM is
  protecting UNIVERSAL .
  protecting TRUTH-VALUE .
  op last-term-undefined : -> Universal .
  op [term] : -> Universal .
  eq [term] = last-term-undefined .
endo

ev (progn (obj_LAST-TERM$install) 'done)

obj NZNAT is
  bsort NzNat
    (obj_NZNAT$is_NzNat_token obj_NZNAT$create_NzNat prin1
     obj_NZNAT$is_NzNat) .
  protecting BOOL .
  op _+_ : NzNat NzNat -> NzNat [assoc comm prec 33] .
  op d : NzNat NzNat -> NzNat [comm] .
  op _*_ : NzNat NzNat -> NzNat [assoc comm idr: 1 prec 31] .
  op quot : NzNat NzNat -> NzNat [gather (E e) prec 31] .
  op _<_ : NzNat NzNat -> Bool [prec 51] .
  op _<=_ : NzNat NzNat -> Bool [prec 51] .
  op _>_ : NzNat NzNat -> Bool [prec 51] .
  op _>=_ : NzNat NzNat -> Bool [prec 51] .
  op s_ : NzNat -> NzNat [prec 15] .
  vars NN NM : NzNat .
  bq NN + NM = (+ NN NM) .
  bq d(NN,NM) = (if (= NN NM) 1 (abs (- NN NM))) .
  bq NN * NM = (* NN NM) .
  bq quot(NN,NM) = (if (> NN NM) (truncate NN NM) 1) .
  bq NN < NM = (< NN NM) .
  bq NN <= NM = (<= NN NM) .
  bq NN > NM = (> NN NM) .
  bq NN >= NM = (>= NN NM) .
  bq s NN = (1+ NN) .
jbo

obj NAT is
  bsort Nat
    (obj_NAT$is_Nat_token obj_NAT$create_Nat prin1
     obj_NAT$is_Nat) .
  protecting NZNAT .
  bsort Zero
    (obj_NAT$is_Zero_token obj_NAT$create_Zero prin1
     obj_NAT$is_Zero) .
  subsorts NzNat < Nat .
  subsorts Zero < Nat .
  op _+_ : Nat Nat -> Nat [assoc comm idr: 0 prec 33] .
  op sd : Nat Nat -> Nat [comm] .
  op _*_ : Nat Nat -> Nat [assoc comm idr: 1 prec 31] .
  op _quo_ : Nat NzNat -> Nat [gather (E e) prec 31] .
  op _rem_ : Nat NzNat -> Nat [gather (E e) prec 31] .
```

```
  op _divides_ : NzNat Nat -> Bool [prec 51] .
  op _<_ : Nat Nat -> Bool [prec 51] .
  op _<=_ : Nat Nat -> Bool [prec 51] .
  op _>_ : Nat Nat -> Bool [prec 51] .
  op _>=_ : Nat Nat -> Bool [prec 51] .
  op s_ : Nat -> NzNat [prec 15] .
  op p_ : NzNat -> Nat [prec 15] .
  var M N : Nat .
  var NN : NzNat .
***  eq N + 0 = N .
  bq sd(M,N) = (abs (- M N)) .
  eq N * 0 = 0 .
  bq M quo NN = (truncate M NN) .
  bq M rem NN = (rem M NN) .
  bq NN divides M = (= 0 (rem M NN)) .
  eq N < 0 = false .
  eq 0 < NN = true .
  eq NN <= 0 = false .
  eq 0 <= N = true .
  eq 0 > N = false .
  eq NN > 0 = true .
  eq 0 >= NN = false .
  eq N >= 0 = true .
  eq s 0 = 1 .
  bq p NN = (- NN 1) .
jbo

obj INT is
  bsort Int
    (obj_INT$is_Int_token obj_INT$create_Int prin1
     obj_INT$is_Int) .
  bsort NzInt
    (obj_INT$is_NzInt_token obj_INT$create_NzInt prin1
     obj_INT$is_NzInt) .
  protecting NAT .
  subsorts Nat < Int .
  subsorts NzNat < NzInt < Int .
  op -_ : Int -> Int [prec 15] .
  op -_ : NzInt -> NzInt [prec 15] .
  op _+_ : Int Int -> Int [assoc comm idr: 0 prec 33] .
  op _-_ : Int Int -> Int [gather (E e) prec 33] .
  op _*_ : Int Int -> Int [assoc comm idr: 1 prec 31] .
  op _*_ : NzInt NzInt -> NzInt [assoc comm idr: 1 prec 31] .
  op _quo_ : Int NzInt -> Int [gather (E e) prec 31] .
  op _rem_ : Int NzInt -> Int [gather (E e) prec 31].
  op _divides_ : NzInt Int -> Bool [prec 51] .
  op _<_ : Int Int -> Bool [prec 51] .
  op _<=_ : Int Int -> Bool [prec 51] .
  op _>_ : Int Int -> Bool [prec 51] .
  op _>=_ : Int Int -> Bool [prec 51] .
  op s_ : Int -> Int [prec 15] .
  vars I J : Int .
  var NJ : NzInt .
  bq - I = (- I) .
  bq I + J = (+ I J) .
***  bq I - J = (- I J) .
  eq I - J = I + (- J) .
  bq I * J = (* I J) .
```

```
  bq I quo NJ = (truncate I NJ) .
  bq I rem NJ = (rem I NJ) .
  bq NJ divides I = (= 0 (rem I NJ)) .
  bq I < J = (< I J) .
  bq I <= J = (<= I J) .
  bq I > J = (> I J) .
  bq I >= J = (>= I J) .
  eq s I = 1 + I .
jbo

obj RAT is
  bsort Rat
    (obj_RAT$is_Rat_token obj_RAT$create_Rat obj_RAT$print
     rationalp) .
  bsort NzRat
    (obj_RAT$is_NzRat_token obj_RAT$create_NzRat obj_RAT$print
     obj_RAT$is_NzRat) .
  protecting INT .
  subsorts Int < Rat .
  subsorts NzInt < NzRat < Rat .
  op -_  : Rat -> Rat [prec 15] .
  op -_  : NzRat -> NzRat [prec 15] .
  op _+_ : Rat Rat -> Rat [assoc comm idr: 0 prec 33] .
  op _-_ : Rat Rat -> Rat [gather (E e) prec 33] .
  op _*_ : Rat Rat -> Rat [assoc comm idr: 1 prec 31] .
  op _*_ : NzRat NzRat -> NzRat [assoc comm idr: 1 prec 31] .
  op _/_ : Rat NzRat -> Rat [gather (E e) prec 31] .
  op _/_ : NzRat NzRat -> NzRat [gather (E e) prec 31] .
  op _rem_ : Rat NzRat -> Rat [gather (E e) prec 31] .
  op _<_ : Rat Rat -> Bool [prec 51] .
  op _<=_ : Rat Rat -> Bool [prec 51] .
  op _>_ : Rat Rat -> Bool [prec 51] .
  op _>=_ : Rat Rat -> Bool [prec 51] .
  vars R S : Rat .
  vars NS : NzRat .
  bq - R = (- R) .
  bq R + S = (+ R S) .
***  bq R - S = (- R S) .
  eq R - S = R + (- S) .
  bq R * S = (* R S) .
  bq R / NS = (/ R NS) .
  bq R rem NS = (rem R NS) .
  bq R < S = (< R S) .
  bq R <= S = (<= R S) .
  bq R > S = (> R S) .
  bq R >= S = (>= R S) .
jbo

obj ID is
 bsort Id (obj_ID$is_Id_token obj_ID$create_Id obj_ID$print_Id
   obj_ID$is_Id) .
 pr BOOL .
 op _<_ : Id Id -> Bool [prec 51] .
 var !X !Y : Id .
 --- the variable names have been chosen so that they are not Id's
 bq !X < !Y = (string< !X !Y) .
endo
```

```
obj QID is
 --- Quoted IDentifier
 --- symbols starting with ' character
 bsort Id (obj_QID$is_Id_token obj_QID$create_Id obj_QID$print_Id
   obj_QID$is_Id) .
endo

obj QIDL is
 protecting QID .
 pr BOOL .
 op _<_ : Id Id -> Bool [prec 51] .
 var X Y : Id .
 bq X < Y = (string< X Y) .
endo

th TRIV is
  sort Elt .
endth

obj FLOAT is
 bsort Float
   (obj_FLOAT$is_Float_token obj_FLOAT$create_Float obj_FLOAT$print_Float
    obj_FLOAT$is_Float) .
 pr BOOL .

 op -_ : Float -> Float [prec 15] .
 op _+_ : Float Float -> Float [assoc comm prec 33] .
 op _-_ : Float Float -> Float [gather (E e) prec 33] .
 op _*_ : Float Float -> Float [assoc comm prec 31] .
 op _/_ : Float Float -> Float [gather (E e) prec 31] .
 op _rem_ : Float Float -> Float [gather (E e) prec 31] .
 op exp : Float -> Float .
 op log : Float -> Float .
 op sqrt : Float -> Float .
 op abs : Float -> Float .
 op sin : Float -> Float .
 op cos : Float -> Float .
 op atan : Float -> Float .
 op pi : -> Float .
 op _<_ : Float Float -> Bool [prec 51] .
 op _<=_ : Float Float -> Bool [prec 51] .
 op _>_ : Float Float -> Bool [prec 51] .
 op _>=_ : Float Float -> Bool [prec 51] .
 op _=[_]_ : Float Float Float -> Bool [prec 51] .

 vars X Y Z : Float .
 bq X + Y = (+ X Y) .
 bq - X = (- X) .
 bq X - Y = (- X Y) .
 bq X * Y = (* X Y) .
 bq X / Y = (/ X Y) .
 bq X rem Y = (rem X Y) .
 bq exp(X) = (exp X) .
 bq log(X) = (log X) .
 bq sqrt(X) = (sqrt X) .
 bq abs(X) = (abs X) .
 bq sin(X) = (sin X) .
 bq cos(X) = (cos X) .
```

```
 bq atan(X) = (atan X) .
 bq pi = pi .
 bq X < Y = (< X Y) .
 bq X <= Y = (<= X Y) .
 bq X > Y = (> X Y) .
 bq X >= Y = (>= X Y) .
 bq (X =[ Z ] Y) = (< (abs (- X Y)) Z) .
endo

obj 2TUPLE[C1 :: TRIV, C2 :: TRIV] is
  sort 2Tuple .
  op <<_;_>> : Elt.C1 Elt.C2 -> 2Tuple .
  op 1*_ : 2Tuple -> Elt.C1 .
  op 2*_ : 2Tuple -> Elt.C2 .
  var e1 : Elt.C1 .
  var e2 : Elt.C2 .
  eq 1* << e1 ; e2 >> = e1 .
  eq 2* << e1 ; e2 >> = e2 .
endo

obj 3TUPLE[C1 :: TRIV, C2 :: TRIV, C3 :: TRIV] is
  sort 3Tuple .
  op <<_;_;_>> : Elt.C1 Elt.C2 Elt.C3 -> 3Tuple .
  op 1*_ : 3Tuple -> Elt.C1 .
  op 2*_ : 3Tuple -> Elt.C2 .
  op 3*_ : 3Tuple -> Elt.C3 .
  var e1 : Elt.C1 .
  var e2 : Elt.C2 .
  var e3 : Elt.C3 .
  eq 1* << e1 ; e2 ; e3 >> = e1 .
  eq 2* << e1 ; e2 ; e3 >> = e2 .
  eq 3* << e1 ; e2 ; e3 >> = e3 .
endo

obj 4TUPLE[C1 :: TRIV, C2 :: TRIV, C3 :: TRIV, C4 :: TRIV] is
  sort 4Tuple .
  op <<_;_;_;_>> : Elt.C1 Elt.C2 Elt.C3 Elt.C4 -> 4Tuple .
  op 1*_ : 4Tuple -> Elt.C1 .
  op 2*_ : 4Tuple -> Elt.C2 .
  op 3*_ : 4Tuple -> Elt.C3 .
  op 4*_ : 4Tuple -> Elt.C4 .
  var e1 : Elt.C1 .
  var e2 : Elt.C2 .
  var e3 : Elt.C3 .
  var e4 : Elt.C4 .
  eq 1* << e1 ; e2 ; e3 ; e4 >> = e1 .
  eq 2* << e1 ; e2 ; e3 ; e4 >> = e2 .
  eq 3* << e1 ; e2 ; e3 ; e4 >> = e3 .
  eq 4* << e1 ; e2 ; e3 ; e4 >> = e4 .
endo

ev (setq *obj$include_BOOL* t)
ev (progn (obj$prelude_install) 'done)
```

References

[1] Hitoshi Aida, Joseph Goguen, Sany Leinwand, Patrick Lincoln, José Meseguer, Babak Taheri, and Timothy Winkler. Simulation and performance estimation for the Rewrite Rule Machine. In *Proceedings, Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 336–344. IEEE, October 1992.

[2] Hitoshi Aida, Joseph Goguen, and José Meseguer. Compiling concurrent rewriting onto the rewrite rule machine. In Stéphane Kaplan and Misuhiro Okada, editors, *Conditional and Typed Rewriting Systems*, pages 320–332. Springer, 1991. Lecture Notes in Computer Science, Volume 516; also, Technical Report SRI-CSL-90-03, Computer Science Lab, SRI International, February, 1990.

[3] Antonio Alencar and Joseph Goguen. OOZE: An object-oriented Z environment. In Pierre America, editor, *European Conference on Object Oriented Programming*. Springer, 1991. Lecture Notes in Computer Science, Volume 512.

[4] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge, 1998.

[5] Eugenio Battiston, Fiorella De Cindio, and Giancarlo Mauri. OBJSA net systems: a class of high-level nets having objects as domains. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, to appear.

[6] Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, Seventh Colloquium*, pages 76–90. Springer, 1980. Lecture Notes in Computer Science, Volume 81.

[7] Rod Burstall. Programming with modules as typed functional programming. *Proceedings, International Conference on Fifth Generation Computing Systems*, 1985.

[8] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[9] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer, 1980. Lecture Notes in Computer Science, Volume 86; based on unpublished notes handed out at the Symposium on Algebra and Applications, Stefan Banach Center, Warsaw, Poland, 1978.

[10] Rod Burstall and Joseph Goguen. An informal introduction to specifications using Clear. In Robert Boyer and J Moore, editors, *The Correctness Problem in Computer Science*, pages 185–213. Academic, 1981. Reprinted in *Software Specification Techniques*, Narain Gehani and Andrew McGettrick, editors, Addison-Wesley, 1985, pages 363–390.

[11] Rod Burstall and Joseph Goguen. Algebras, theories and freeness: An introduction for computer scientists. In Martin Wirsing and Gunther Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 329–350. Reidel, 1982. Proceedings, 1981 Marktoberdorf NATO Summer School, NATO Advanced Study Institute Series, Volume C91.

[12] Rod Burstall and Butler Lampson. A kernel language for abstract data types and modules. In Giles Kahn, David MacQueen, and Gordon Plotkin, editors, *Proceedings, International Symposium on the Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 1984.

[13] Rod Burstall, David MacQueen, and Donald Sannella. Hope: an experimental applicative language. In *Proceedings, First LISP Conference*, volume 1, pages 136–143. Stanford University, 1980.

[14] Carlo Cavenathi, Marco De Zanet, and Giancarlo Mauri. MC-OBJ: a C interpreter for OBJ. *Note di Software*, 36/37:16–26, October 1988. In Italian.

[15] Thomas Cheatham. The introduction of definitional facilities into higher level programming languages. In *Proceedings, AFIPS Fall Joint Computer Conference*, pages 623–637. Spartan Books, 1966.

[16] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In José Meseguer, editor, *Proceedings, First International Workshop on Rewriting Logic and its Applications*. Elsevier Science, 1996. Volume 4, *Electronic Notes in Theoretical Computer Science*.

[17] Manuel Clavel, Steven Eker, and José Meseguer. Current design and implementation of the Cafe prover and Knuth-Bendix tools, 1997. Presented at CafeOBJ Workshop, Kanazawa, October 1997.

[18] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In José Meseguer, editor, *Proceedings, First International Workshop on Rewriting Logic and its Applications*. Elsevier Science, 1996. Volume 4, *Electronic Notes in Theoretical Computer Science*.

[19] CoFI. CASL summary, 1999. `http://www.brics.dk/Projects/CoFi/`.

[20] Derek Coleman, Robin Gallimore, and Victoria Stavridou. The design of a rewrite rule interpreter from algebraic specifications. *IEE Software Engineering Journal*, July:95–104, 1987.

[21] Hubert Comon. *Unification et Disunification: Théories et Applications*. PhD thesis, Université de l'Institut Polytechnique de Grenoble, 1988.

[22] Department of Defense. Reference manual for the Ada programming language. United States Government, Report ANSI/MIL-STD-1815A, 1983.

[23] Nachum Dershowitz and Jean-Pierre Jouannaud. Notations for rewriting. *Bulletin of the European Association for Theoretical Computer Science*, 43:162–172, 1990.

[24] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewriting systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Methods and Semantics*, pages 243–320. North-Holland, 1990.

[25] Răzvan Diaconescu. *Category-based Semantics for Equational and Constraint Logic Programming*. PhD thesis, Programming Research Group, Oxford University, 1994.

[26] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific, 1998. AMAST Series in Computing, Volume 6.

[27] Răzvan Diaconescu and Kokichi Futatsugi. Logical foundations of CafeOBJ, 1999. Submitted for publication.

[28] David Duce. Concerning the compatibility of PHIGS and GKS. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, to appear.

[29] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, 1985. EATCS Monographs on Theoretical Computer Science, Volume 6.

[30] Kokichi Futatsugi. Hierarchical software development in HISP. In T. Kitagawa, editor, *Computer Science and Technologies 1982*, pages 151–174. OHMSA/North Holland, 1982. Japan Annual Review in Electronics, Computer and Telecommunications Series.

[31] Kokichi Futatsugi. An overview of OBJ2. In Kazuhiru Fuchi and Maurice Nivat, editors, *Proceedings, France-Japan AI and CS Symposium*. ICOT, 1986. Also Information Processing Society of Japan, Technical Memorandum PL-86-6.

[32] Kokichi Futatsugi. Trends in formal specification methods based on algebraic specification techniques — from abstract data types to software processes: A personal perspective. In *Proceedings, International Conference Commemorating the 30th Anniversary of the Information Processing Society of Japan*, pages 59–66. Information Processing Society of Japan, 1990.

[33] Kokichi Futatsugi. Product-centered process description = algebraic specification of environment + SCRIPT. In *Proceedings, 6th International Software Process Workshop*, pages 95–97. IEEE, 1991.

[34] Kokichi Futatsugi. Structuring and derivation in algebraic specification/programming languages. *Journal of Information Processing Society of Japan*, 14(2):153–163, 1991.

[35] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings, Twelfth ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[36] Kokichi Futatsugi, Joseph Goguen, José Meseguer, and Koji Okada. Parameterized programming in OBJ2. In Robert Balzer, editor, *Proceedings, Ninth International Conference on Software Engineering*, pages 51–60. IEEE Computer Society, March 1987.

[37] Kokichi Futatsugi, Joseph Goguen, José Meseguer, and Koji Okada. Parameterized programming and its application to rapid prototyping in OBJ2. In Yoshihiro Matsumoto and Yutaka Ohno, editors, *Japanese Perspectives on Software Engineering*, pages 77–102. Addison Wesley, 1989.

[38] Kokichi Futatsugi and Koji Okada. Specification writing as construction of hierarchically structured clusters of operators. In *Proceedings, 1980 IFIP Congress*, pages 287–292. IFIP, 1980.

[39] Kokichi Futatsugi and Koji Okada. A hierarchical structurting method for functional software systems. In *Proceedings, 6th International Conference on Software Engineering*, pages 393–402. IEEE, 1982.

[40] Christopher Paul Gerrard. The specification and controlled implementation of a configuration management tool using OBJ and Ada, 1988. Gerrard Software Limited.

[41] Martin Gogolla. Partially ordered sorts in algebraic specifications. In Bruno Courcelle, editor, *Proceedings, Ninth CAAP (Bordeaux)*, pages 139–153. Cambridge, 1984. Also Forschungsbericht Nr. 169, Universität Dortmund, Abteilung Informatik, 1983.

[42] Martin Gogolla. A final algebra semantics for errors and exceptions. In Hans-Jörg Kreowski, editor, *Recent Trends in Data Type Specification*, volume Informatik-Fachberichte 116, pages 89–103. Springer, 1985. Selected papers from the Third Workshop on Theory and Applications of Abstract Data Types.

[43] Joseph Goguen. Abstract errors for abstract data types. In Eric Neuhold, editor, *Proceedings, First IFIP Working Conference on Formal Description of Programming Concepts*, pages 21.1–21.32. MIT, 1977. Also in *Formal Description of Programming Concepts*, Peter Neuhold, Ed., North-Holland, pages 491–522, 1979.

[44] Joseph Goguen. Order sorted algebra. Technical Report 14, UCLA Computer Science Department, 1978. Semantics and Theory of Computation Series.

[45] Joseph Goguen. Some design principles and theory for OBJ-0, a language for expressing and executing algebraic specifications of programs. In Edward Blum, Manfred Paul, and Satsoru Takasu, editors, *Proceedings, Conference on Mathematical Studies of Information Processing*, pages 425–473. Springer, 1979. Lecture Notes in Computer Science, Volume 75.

[46] Joseph Goguen. How to prove algebraic inductive hypotheses without induction, with applications to the correctness of data type representations. In Wolfgang Bibel and Robert Kowalski, editors, *Proceedings, Fifth Conference on Automated Deduction*, pages 356–373. Springer, 1980. Lecture Notes in Computer Science, Volume 87.

[47] Joseph Goguen. Parameterized programming. *Transactions on Software Engineering*, SE–10(5):528–543, September 1984.

[48] Joseph Goguen. Merged views, closed worlds and ordered sorts: Some novel database features in OBJ. In Alex Borgida and Peter Buneman, editors, *Workshop on Database Interfaces*, pages 38–47. University of Pennsylvania, Computer Science Department, 1985. This workshop took place in October, 1982.

[49] Joseph Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, February 1986. Reprinted in *Tutorial: Software Reusability*, Peter Freeman, editor, IEEE Computer Society, 1987, pages 251–263, and in *Domain Analysis and Software Systems Modelling*, Rubén Prieto-Díaz and Guillermo Arango, editors, IEEE Computer Society, 1991, pages 125–137.

[50] Joseph Goguen. Modular algebraic specification of some basic geometrical constructions. *Artificial Intelligence*, pages 123–153, 1988. Special Issue on Computational Geometry, edited by Deepak Kapur and Joseph Mundy; also, Report CSLI-87-87, Center for the Study of Language and Information at Stanford University, March 1987.

[51] Joseph Goguen. OBJ as a theorem prover, with application to hardware verification. In V.P. Subramanyan and Graham Birtwhistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 218–267. Springer, 1989.

[52] Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.

[53] Joseph Goguen. Proving and rewriting. In Hélène Kirchner and Wolfgang Wechler, editors, *Proceedings, Second International Conference on Algebraic and Logic Programming*, pages 1–24. Springer, 1990. Lecture Notes in Computer Science, Volume 463.

[54] Joseph Goguen. Semantic specifications for the Rewrite Rule Machine. In Aki Yonezawa and Takayasu Ito, editors, *Concurrency: Theory, Language and Architecture*, pages 216–234, 1991. Proceedings of a U.K.–Japan Workshop; Springer, Lecture Notes in Computer Science, Volume 491.

[55] Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.

[56] Joseph Goguen. Stretching first order equational logic: Proofs with partiality, subtypes and retracts. In Maria Paola Bonacina and Ulrich Furbach, editors, *Proceedings, Workshop on First Order Theorem Proving*, pages 78–85. Johannes Kepler Univ. Linz, 1997. Schloss Hagenberg, Austria, October 1997; RISC-Linz Report No. 95–50; revised version to appear in *Journal of Symbolic Computation*.

[57] Joseph Goguen. Cse 230 homepage, 1999. `http://www.cs.ucsd.edu/users/goguen/courses/230`.

[58] Joseph Goguen. An introduction to algebraic semiotics, with applications to user interface design. In Chrystopher Nehaniv, editor, *Computation for Metaphors, Analogy and Agents*, pages 242–291. Springer, 1999. Lecture Notes in Artificial Intelligence, Volume 1562.

[59] Joseph Goguen. *Theorem Proving and Algebra*. MIT, to appear.

[60] Joseph Goguen. Social and semiotic analyses for theorem prover user interface design, to appear, 2000. Special issue on user interfaces for theorem provers.

[61] Joseph Goguen and Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994.

[62] Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Operational semantics of order-sorted algebra. In Wilfried Brauer, editor, *Proceedings, 1985 International Conference on Automata, Languages and Programming*. Springer, 1985. Lecture Notes in Computer Science, Volume 194.

[63] Joseph Goguen, Claude Kirchner, José Meseguer, and Timothy Winkler. OBJ as a language for concurrent programming. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, Second International Supercomputing Conference, Volume I*, pages 195–198. International Supercomputing Institute Inc. (St. Petersburg FL), 1987.

[64] Joseph Goguen, Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Distributed cooperative formal methods tools. In Michael Lowry, editor, *Proceedings, Automated Software Engineering*, pages 55–62. IEEE, 1997.

[65] Joseph Goguen, Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Tools for distributed cooperative design and validation. In *Proceedings, CafeOBJ Symposium*. Japan Advanced Institute for Science and Technology, 1998. Namazu, Japan, April 1998.

[66] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT, 1996.

[67] Joseph Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97–538, UCSD, Dept. Computer Science & Eng., May 1997. To appear in special issue of *Theoretical Computer Science* on Algebraic Engineering, edited by Chrystopher Nehaniv and Masamo Ito. Extended abstract in *Proc., Conf. Intelligent Systems: A Semiotic Perspective, Vol. I*, ed. J. Albus, A. Meystel and R. Quintero, Nat. Inst. Science & Technology (Gaithersberg MD, 20–23 October 1996), pages 159–167.

[68] Joseph Goguen and Grant Malcolm. Hidden coinduction: Behavioral correctness proofs for objects. *Mathematical Structures in Computer Science*, 9(3):287–319, June 1999.

[69] Joseph Goguen and José Meseguer. Rapid prototyping in the OBJ executable specification language. *Software Engineering Notes*, 7(5):75–84, December 1982. Proceedings of Rapid Prototyping Workshop.

[70] Joseph Goguen and José Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In M. Nielsen and E.M. Schmidt, editors, *Proceedings, 9th International Conference on Automata, Languages and Programming*, pages 265–281. Springer, 1982. Lecture Notes in Computer Science, Volume 140.

[71] Joseph Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179–210, September 1984.

[72] Joseph Goguen and José Meseguer. Models and equality for logical programming. In Hartmut Ehrig, Giorgio Levi, Robert Kowalski, and Ugo Montanari, editors, *Proceedings, 1987 TAPSOFT*, pages 1–22. Springer, 1987. Lecture Notes in Computer Science, Volume 250.

[73] Joseph Goguen and José Meseguer. Order-sorted algebra solves the constructor selector, multiple representation and coercion problems. In *Proceedings, Second Symposium on Logic in Computer Science*, pages 18–29. IEEE Computer Society, 1987. Also Report CSLI-87-92, Center for the Study of Language and Information, Stanford University, March 1987; revised version in *Information and Computation, 103*, 1993.

[74] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153–162, October 1986.

[75] Joseph Goguen and José Meseguer. Software for the Rewrite Rule Machine. In Hideo Aiso and Kazuhiro Fuchi, editors, *Proceedings, International Conference on Fifth Generation Computer Systems 1988*, pages 628–637. Institute for New Generation Computer Technology (ICOT), 1988.

[76] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992. Also Programming Research Group Technical Monograph PRG–80, Oxford University, December 1989, and Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab, July 1989; originally given as lecture at *Seminar on Types*, Carnegie-Mellon University, June 1983; many draft versions exist, from as early as 1985.

[77] Joseph Goguen, José Meseguer, and David Plaisted. Programming with parameterized abstract objects in OBJ. In Domenico Ferrari, Mario Bolognani, and Joseph Goguen, editors, *Theory and Practice of Software Technology*, pages 163–193. North-Holland, 1983.

[78] Joseph Goguen, Akira Mori, and Kai Lin. Algebraic semiotics, ProofWebs and distributed cooperative proving. In Yves Bartot, editor, *Proceedings, User Interfaces for Theorem Provers*, pages 25–34. INRIA, 1997. (Sophia Antipolis, 1–2 September 1997).

[79] Joseph Goguen and Kamran Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In J. Diaz and I. Ramos, editors, *Formalizing Programming Concepts*, pages 292–309. Springer, 1981. Lecture Notes in Computer Science, Volume 107.

[80] Joseph Goguen and Grigore Roşu. Hiding more of hidden algebra. In *Proceedings, World Congress on Formal Methods, 1999.* Springer, to appear 1999. Toulouse, France.

[81] Joseph Goguen, Andrew Stevens, Keith Hobley, and Hendrik Hilberdink. 2OBJ, a metalogical framework based on equational logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992. Also in *Mechanized Reasoning and Hardware Design*, edited by C.A.R. Hoare and Michael J.C. Gordon, Prentice-Hall, 1992, pages 69–86.

[82] Joseph Goguen and Joseph Tardo. An introduction to OBJ: A language for writing and testing software specifications. In Marvin Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE, 1979. Reprinted in *Software Specification Techniques*, Nehan Gehani and Andrew McGettrick, editors, Addison Wesley, 1985, pages 391–420.

[83] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology, IV*, pages 80–149. Prentice-Hall, 1978.

[84] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Abstract data types as initial algebras and the correctness of data representations. In Alan Klinger, editor, *Computer Graphics, Pattern Recognition and Data Structure*, pages 89–93. IEEE, 1975.

[85] Joseph Goguen and William Tracz. An implementation-oriented semantics for module composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component-based Systems.* Cambridge, 1999. To appear.

[86] Joseph Goguen and Timothy Winkler. Introducing OBJ3. Technical report, SRI International, Computer Science Lab, 1988.

[87] Joseph Goguen and David Wolfram. On types and FOOPS. In Robert Meersman, William Kent, and Samit Khosla, editors, *Object Oriented Databases: Analysis, Design and Construction*, pages 1–22. North Holland, 1991. Proceedings, IFIP TC2 Conference, Windermere, UK, 2–6 July 1990.

[88] Robert Goldblatt. *Topoi, the Categorial Analysis of Logic.* North-Holland, 1979.

[89] Michael J.C. Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF.* Springer, 1979. Lecture Notes in Computer Science, Volume 78.

[90] John Guttag. *The Specification and Application to Programming of Abstract Data Types.* PhD thesis, University of Toronto, 1975. Computer Science Department, Report CSRG–59.

[91] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, 1986.

[92] Jieh Hsiang. *Refutational Theorem Proving using Term Rewriting Systems.* PhD thesis, University of Illinois at Champaign-Urbana, 1981.

[93] Gérard Huet and Derek Oppen. Equations and rewrite rules: A survey. In Ron Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic, 1980.

[94] John Hughes. Why functional programming matters. Technical Report 16, Programming Methodology Group, University of Goteborg, November 1984.

[95] John Hughes. Abstract interpretations of first order polymorphic functions. In Cordelia Hall, John Hughes, and John O'Donnell, editors, *Proceedings of the 1988 Glasgow Workshop on Functional Programming*, pages 68–86. Computing Science Department, University of Glasgow, 1989.

[96] Jean-Pierre Jouannaud and Claude Marché. Completion modulo associativity, commutativity and identity. In Alfonso Miola, editor, *Proceedings, DISCO '90*, pages 111–120. Springer, 1991. Lecture Notes in Computer Science, Volume 429; to appear in *Theoretical Computer Science.*

[97] Jean-Pierre Jouannaud and Mitsuhiro Okada. Executable higher-order algebraic specification languages. In *Proceedings, 6th Symposium on Logic in Computer Science*, pages 350–361. IEEE, 1991.

[98] Heinz Kaphengst and Horst Reichel. Initial algebraic semantics for non-context-free languages. In Marek Karpinski, editor, *Fundamentals of Computation Theory*, pages 120–126. Springer, 1977. Lecture Notes in Computer Science, Volume 56.

[99] Claude Kirchner. Order sorted equational matching, 1988.

[100] Claude Kirchner, Hélène Kirchner, and Aristide Mégrelis. OBJ for OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action.* Kluwer, to appear.

[101] Claude Kirchner, Hélène Kirchner, and José Meseguer. Operational semantics of OBJ3. In T. Lepistö and Aarturo Salomaa, editors, *Proceedings, 15th International Colloquium on Automata, Languages and Programming*, pages 287–301. Springer, 1988. (Tampere, Finland, 11-15 July 1988.) Lecture Notes in Computer Science, Volume 317.

[102] Jan Willem Klop. Term rewriting systems: A tutorial. *Bulletin of the European Association for Theoretical Computer Science*, 32:143–182, June 1987.

[103] Jan Willem Klop. Term rewriting systems: from Church-Rosser to Knuth-Bendix and beyond. In Samson Abramsky, Dov Gabbay, and Tom Maibaum, editors, *Handbook of Logic in Computer Science*, pages 1–117. Oxford, 1992.

[104] Donald Knuth and Peter Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*. Pergamon, 1970.

[105] John T. Latham. Abstract Pascal: A tutorial introduction. Technical Report Version 2.1, University of Manchester, Department of Computer Science, 1987.

[106] Sany Leinwand and Joseph Goguen. Architectural options for the Rewrite Rule Machine. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, Second International Supercomputing Conference, Volume I*, pages 63–70. International Supercomputing Institute Inc. (St. Petersburg FL), 1987.

[107] Sany Leinwand, Joseph Goguen, and Timothy Winkler. Cell and ensemble architecture of the Rewrite Rule Machine. In Hideo Aiso and Kazuhiro Fuchi, editors, *Proceedings, International Conference on Fifth Generation Computer Systems 1988*, pages 869–878. Institute for New Generation Computer Technology (ICOT), 1988.

[108] Patrick Lincoln and Jim Christian. Adventures in associative-commutative unification. *Journal of Symbolic Computation*, 8:217–240, 1989. Also appears in *Unification*, edited by Claude Kirchner (Academic, 1990), pages 393–416.

[109] Grant Malcolm and Joseph Goguen. An executable course on the algebraic semantics of imperative programs. In Michael Hinchey and C. Neville Dean, editors, *Teaching and Learning Formal Methods*, pages 161–179. Academic, 1996.

[110] Narciso Martí-Oliet and José Meseguer. Inclusions and subtypes. Technical Report SRI-CSL-90-16, SRI International, Computer Science Lab, December 1990.

[111] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holand, 1982.

[112] José Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM, 1990.

[113] José Meseguer. Conditional rewriting logic: Deduction, models and concurrency. In Stéphane Kaplan and Misuhiro Okada, editors, *Conditional and Typed Rewriting Systems*, pages 64–91. Springer, 1991. Lecture Notes in Computer Science, Volume 516.

[114] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[115] José Meseguer. Membership algebra as a logical framework for equational specification, 1997. Draft manuscript. Computer Science Lab, SRI International.

[116] José Meseguer and Joseph Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.

[117] José Meseguer, Joseph Goguen, and Gert Smolka. Order-sorted unification. *Journal of Symbolic Computation*, 8:383–413, 1989.

[118] Donald Michie. 'Memo' functions and machine learning. *Nature*, 218:19–22, April 1968.

[119] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[120] Peter Mosses. Abstract semantic algebras! In Dines Bjorner, editor, *Formal Description of Programming Concepts II*, pages 45–70. IFIP, 1983.

[121] Peter Mosses. A basic semantic algebra. In Giles Kahn, David MacQueen, and Gordon Plotkin, editors, *Proceedings, International Symposium on the Semantics of Data Types*, pages 87–107. Springer, 1985. Lecture Notes in Computer Science, Volume 173.

[122] Peter Mosses. Unified algebras and institutions. In *Proceedings, Fourth Annual Conference on Logic in Computer Science*, pages 304–312. IEEE, 1989.

[123] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edingurgh, 1981.

[124] Ataru Nakagawa and Kokichi Futatsugi. Stepwise refinement process with modularity: An algebraic approach. In *Proceedings, 11th International Conference on Software Engineering*, pages 166–177. IEEE, 1989.

[125] Ataru Nakagawa and Kokichi Futatsugi. Software process *a la* algebra: OBJ for OBJ. In *Proceedings, 12th International Conference on Software Engineering*, pages 12–32. IEEE, 1990.

[126] Ataru Nakagawa, Kokichi Futatsugi, S. Tomura, and T. Shimizu. Algebraic specification of MacIntosh's QuickDraw using OBJ2. Technical Report Draft, ElectroTechnical Laboratory, Tsukuba Science City, Japan, 1987. In *Proceedings*, Tenth International Conference on Software Engineering, Singapore, April 1988.

[127] Kazuhito Ohmaki, Kokichi Futatsugi, and Koichi Takahashi. A basic LOTOS simulator in OBJ. In *Proceedings, International Conference Commemorating the 30th Anniversary of the Information Processing Society of Japan*, pages 497–504. Information Processing Society of Japan, 1990.

[128] Kazuhito Ohmaki, Koichi Takahashi, and Kokichi Futatsugi. A LOTOS simulator in OBJ. In *Proceedings, FORTE'90: Third International Conference on Formal Description Techniques*, November 1990.

[129] Lawrence Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge, 1987. Cambridge Tracts in Theoretical Computer Science, Volume 2.

[130] Francisco Pinheiro and Joseph Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, pages 52–64, March 1996. Special issue of papers from ICRE'96.

[131] David Plaisted. An initial algebra semantics for error presentations. SRI International, Computer Science Laboratory, 1982.

[132] Axel Poigné. On semantic algebras: Higher order structures. Informatik II, Universität Dortmund, 1983.

[133] Axel Poigné. Once more on order-sorted algebra. Technical Report Draft, GMD, 1990.

[134] Axel Poigné. Parameterization for order-sorted algebraic specification. *Journal of Computer and System Sciences*, 40(3):229–268, 1990.

[135] John Reynolds. Using category theory to design implicit conversions and generic operators. In Neal Jones, editor, *Semantics Directed Compiler Generation*, pages 211–258. Springer, 1980. Lecture Notes in Computer Science, Volume 94.

[136] David Rydeheard and Rod Burstall. *Computational Category Theory*. Prentice-Hall, 1988.

[137] Augusto Sampaio. A comparative study of theorem provers: Proving correctness of compiling specifications. Technical Report PRG-TR-20-90, Oxford University Computing Laboratory, 1990.

[138] Augusto Sampaio. *An Algebraic Approach to Compiler Design*, volume 4. World Scientific, 1998.

[139] Dana Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In *Proceedings, 21st Symposium on Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn, 1971. Also Programming Research Group Technical Monograph PRG–6, Oxford.

[140] Gert Smolka and Hassan Aït-Kaci. Inheritance hierarchies: Semantics and unification. Technical Report Report AI-057-87, MCC, 1987. In *Journal of Symbolic Computation*, 1988.

[141] Gert Smolka, Werner Nutt, Joseph Goguen, and José Meseguer. Order-sorted equational computation. In Maurice Nivat and Hassan Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, pages 299–367. Academic, 1989. Preliminary version in *Proceedings*, Colloquium on the Resolution of Equations in Algebraic Structures, held in Lakeway, Texas, May 1987, and SEKI Report SR-87-14, Universität Kaiserslautern, December 1987.

[142] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.

[143] S. Sridhar. An implementation of OBJ2: An object-oriented language for abstract program specification. In K.V. Nori, editor, *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 81–95. Springer, 1986. Lecture Notes in Computer Science, Volume 241.

[144] Victoria Stavridou. Specifying in OBJ, verifying in REVE, and some ideas about time. Technical report, Department of Computer Science, University of Manchester, 1987.

[145] Victoria Stavridou, Joseph Goguen, Steven Eker, and Serge Aloneftis. FUNNEL: A CHDL with formal semantics. In *Proceedings, Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 117–144. IEEE, 1991.

[146] Victoria Stavridou, Joseph Goguen, Andrew Stevens, Steven Eker, Serge Aloneftis, and Keith Hobley. FUNNEL and 2OBJ: towards an integrated hardware design environment. In *Theorem Provers in Circuit Design*, volume IFIP Transactions, A-10, pages 197–223. North-Holland, 1992.

[147] Christopher Strachey. Fundamental concepts in programming languages. Lecture Notes from International Summer School in Computer Programming, Copenhagen, 1967.

[148] Joseph Tardo. *The Design, Specification and Implementation of OBJT: A Language for Writing and Testing Abstract Algebraic Program Specifications.* PhD thesis, UCLA, Computer Science Department, 1981.

[149] David Turner. Miranda: A non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architectures*, pages 1–16. Springer, 1985. Lecture Notes in Computer Science, Volume 201.

[150] William Wadge. Classified algebras. Technical Report 46, University of Warwick, October 1982.

[151] Mitchell Wand. First-order identities as a defining language. *Acta Informatica*, 14:337–357, 1980. Originally Report 29, Computer Science Deptartment, Indiana University, 1977.

[152] Timothy Winkler, Sany Leinwand, and Joseph Goguen. Simulation of concurrent term rewriting. In Steven Kartashev and Svetlana Kartashev, editors, *Proceedings, Second International Supercomputing Conference, Volume I*, pages 199–208. International Supercomputing Institute Inc. (St. Petersburg FL), 1987.

[153] Steven Zilles. Abstract specification of data types. Technical Report 119, Computation Structures Group, Massachusetts Institute of Technology, 1974.

Contents